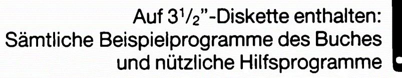
AMIGA

P. Lukowicz/O. Pfeiffer

GRAFIK IN C







AMIGA

P. Lukowicz/O. Pfeiffer

GRAFIK IN C





Grafik in C auf dem Amiga
Lukowicz/Pfeiffer
— 1. Auflage — Darmstadt: Heim, 1990
ISBN 3-923250-91-6

Druck: Druckerei der Heim OHG, 6100 Darmstadt.

Inhaltsverzeichnis

Inhalt

Inhalt

Vorwort

Schreibt man ein Grafikbuch über den Amiga, so bleibt einem an dieser Selle wohl kaum etwas anderes übrig, als die besonderen Grafikleistungen des Amigas hervorzuheben. Dabei wollen wir es dann aber auch belassen, da Ihnen ja wohl die theoretischen Grafikfähigkeiten Ihres Computers bekannt sein dürften, und sei es nur aus irgendwelchen Spielen, Grafikdemos oder auch Malprogrammen. Sobald man aber die Grafikmöglichkeiten durch eigene Programme nutzen möchte, stellt man sehr schnell fest, wie unergründlich die Wege durch den Amiga sind. Nach endlosen Compiler-Durchläufen und bei der dazugehörenden Fehlersuche endet man schließlich in einem Zustand, bei dem jede "Guru Meditation" einen in Trance versetzten kann. Solche Erfahrungen möchten wir Ihnen, zumindest im Bereich der Grafikprogrammierung, durch dieses Buch gerne ersparen. Dabei sei jedoch gesagt, daß bei aktiver Amiga-Programmierung niemand vor solchen Erlebnissen sicher ist, auch mit noch sovielen guten Büchern nicht. Man kann allerdings nicht leugnen, daß gerade in diesem Fall, Bücher über die Systemprogrammierung einen unschätzbaren Wert darstellen. So erfreut es einen dann auch, wenn gerade dieser recht trockene Stoff bereits im Orginal nicht todernst behandelt wird. Beschreibungen wie die der Fehlerwarnung der **SizeWindow**-Routine (siehe Anhang C) geben einem durchaus wieder neue Hoffnung, zumindest bis zu dem Zeitpunkt wo der Amiga wiedereinmal seine spiritistischen Neigungen rausläßt. Alles in allem hat uns das Schreiben dieses Buches jedoch eine Menge gebracht: Spaß, Ärger, weniger Schlaf, einen größeren Kalorienverbrauch in der Zeit zwischen 22 und 2 Uhr, sowie eine gehörige Portion Selbstbeherrschung. Uns

bleibt abschließend nichts weiter übrig, als Ihnen bei den nun folgenden Sitzungen mit Ihrem Amiga viel Spaß und möglichst wenig "Gurus" zu wünschen.

Einleitung und Leseanleitung

Wir möchten Ihnen hier zunächst ein paar Richtlinien vermitteln, die Ihnen beim Verständniß der einzelnen Abschnitte von Nutzen sein werden. Hält man über Computer etwas schriftlich fest so hat man grundsätzlich zwei Möglichkeiten: entweder man übersetzt konsequent Begriffe ins Deutsche, oder man verwendet Sie direkt und geht davon aus, daß jeder weiß was gemeint ist.

Diese beiden, von den Verlagshäusern unterschiedlich gehandhabten "Philosophien", haben sicherlich Vor- und Nachteile, die wir jedoch garnicht erst versucht haben gegeneinander aufzuwiegen. Wir haben uns ohne zögern für den umgangssprachlichen Ton entschieden, der die Amiga-spezifischen Begriffe "natürlich" in ihrem Orginal beläßt. Bei der Systemnahen Programmierung ist dies besonders sinnvoll, da dort Begriffe und Parameter auftauchen, die Sie auch bei Ihrem C-Compiler wiederfinden. Dabei seien bereits hier die Include-Files bemerkt, die Sie bei ihrem C-Compiler in dem Directory "include" finden. Dort sind sämtliche Routinen und Datenstrukturen des Amiga-Betriebssytems definiert und zwar "natürlich" in Englisch.

Da dieses Buch keine Einführung in die Programmiersprache C sein soll, setzen wir bei Ihnen ein Minimum an Amiga-Wissen voraus, daß Begriffe wie "Icon", "Pixel" und "Screen" beinhalted. Umgangssprachlich bedeutet aber auch, daß wir Begriffe wie "Intuition" oder "RastPort" verschieden verwenden. Reden wir von

dem RastPort oder dem ViewPort, so meinen wir in erster Linie die **RastPort**-Datenstruktur, bzw. die **ViewPort**-Datenstruktur. Ein Zeiger auf einen solchen "Port" ist dann logischerweise ein Adreßzeiger auf die zugehörige Datenstruktur. Sagen Ihnen auch diese Begriffe noch nichts, dann sollten Sie in diesem Buch keine Abschnitte oder Kapitel überspringen, da Begriffe nur bei der Einführung erklärt werden. Die Abschnitte und Kapitel bauen aufeinander auf, so daß wir nach der Einführung von Begriffen meißtens bei der kürzeren, umgangssprachlichen Form bleiben. So kommt es, daß Sie Worte wie "Bildschirmwiederholspeicherbereich" in diesem Buch nicht finden werden, sieht man mal von dieser Stelle hier ab.

Kapitel 1

Grundlagen der Grafikdarstellung

In diesem Kapitel geht es um die Grafikdarstellung auf dem Bildschirm und um die Grafikmöglichkeiten des Amiga. Es wird nicht nur auf die einzelnen Grafikmodi LoRes, HighRes, Interlace, Extra-Halfbright und Hold-And-Modify, sowie die Playfields eingegangen, sondern auch auf die Grafikhardware, also z.B. den Coprozessor Blitter, der bei der Grafikprogrammierung eine wichtige Rolle spielt. In diesem Zusammenhang werden auch die Verschiedenen Arten des Speichers: CHIP- und FAST-Memory erläutert.

1. Aufbau des Grafikbildschirms

In diesem Abschnitt geht es um die Unterschiede in den Qualitäten der Bildschirmausgabe und um Begriffe wie Pixel, RGB, Bitplane und BitMap.

Bei der Bildschirmausgabe eines Computers trifft man auf die unterschiedlichsten Qualitäten und Darstellungsmöglichkeiten der verschiedenen Systeme. Im Grunde arbeiten sie jedoch alle nach dem gleichen Prinzip. Das Bild setzt sich aus vielen kleinen Punkten, genannt Pixels, zusammen. Die Qualitätsunterschiede verschiedener Systeme und/oder Grafikmodi entstehen im wesentlichen durch drei Faktoren:

I. Die Auflösung, also aus wievielen Punkten sich das Bild zusammensetzt.

II. Die Intensität und Farbmöglichkeiten jedes einzelnen Pixels, also wieviel verschiedene Helligkeits und Farbstufen jeder Pixel annehmen kann.

III. Die Frequenz, mit der die Bilder aufgebaut wer-

den, also wieviel Bilder letztendlich pro Sekunde auf dem Bildschirm erscheinen.

Die Auflösung ist verantwortlich für die Schärfe des Bildes, Intensität und Farbmöglichkeiten für Farbabstufungen und fließende Farbübergänge, sowie schließlich die Frequenz für die flimmerfreie Darstellung auf dem Bildschirm.

Computerintern sieht das ganze ungefähr so aus: Jedem Pixel ist genau ein Bit im Speicher des Computers zugeordnet. Ist es gesetzt (1), so ist der Pixel an, andernfalls (0) aus. Der Bildschirm ist eine Zusammensetzung aus "Einser" und "Nullen", die die Bitplane ergeben. Bildhaft gesehen ist die Bitplane eine Fläche von Nullen und Einsen. Damit hätten wir jedoch lediglich die Möglichkeit, einen Punkt auf dem Bildschirm zu setzen oder nicht. Um die Farbe der Pixels zu beeinflussen, braucht man jedoch mehr Information, als ein Bit. Dazu legt der Amiga mehrere Bitplanes an, die sich "überlappen". Bei zwei Bitplanes setzt sich die Information für ein Pixel aus einem Bit aus der ersten Bitplane und einem Bit aus der zweiten Bitplane zusammen. Somit kann man jedem Pixel bereits vier Farben zuordnen, eben die Binärkombinationen 00, 01, 10 und 11. Durch hinzufügen weiterer Bitplanes, kann man jedem Pixel 2 hoch (Anzahl an Bitplanes) Farben über die Farbregister zuordnen.

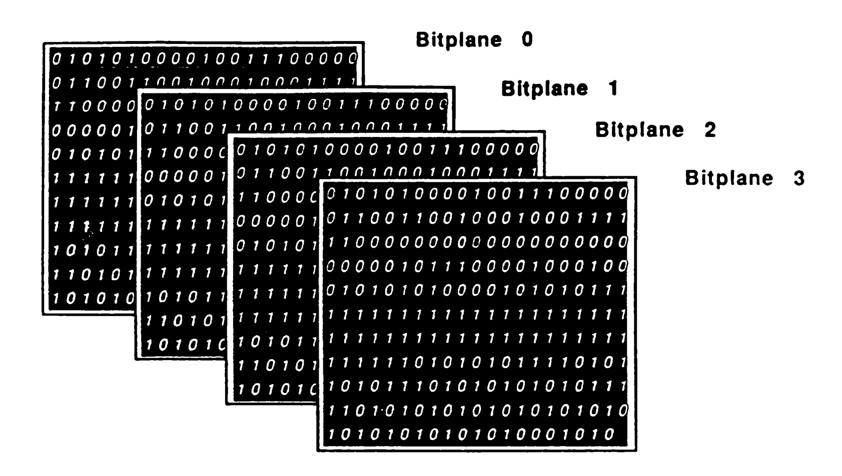


BILD 1.1.: Farbdarstellung durch Bitplanes

In den Farbregistern steht dann der eigentliche Farbwert (einer von 4096), den das Farbsignal als RGB-Signal an den Monitor leitet. Dieses Farbsignal setzt sich aus den Grundfarben **R**ot, **G**rün und **B**lau, eben RGB, zusammen. Das Benutzen mehrerer Bitplanes und deren "Überlappen" nennt man Bitmapping.

2. Die Grafikmodi des Amiga

In Bezug auf die Qualität der Bildschirmausgabe stehen uns beim Amiga mehrere Möglichkeiten, sprich Grafikmodi, zur Verfügung. Aus den Preferences der Workbench wissen Sie, daß die Farbpalette des Amiga sich aus den Farben Rot, Grün und Blau zusammensetzt, die in verschiedenen Abstufungen miteinander gemischt werden können. Da es für jede der drei Farben 16 Abstufungen gibt, stehen uns also insgesamt 4096 (16 hoch drei) Farben zur Verfügung. Was die Auflösung betrifft,so ist der Amiga ebenfalls bestens ausgerüstet, wobei Amiga nicht gleich Amiga ist. Die amerikanische Version verfügt lediglich über maximal 400 Zeilen, während sein deutscher Bruder bis zu 512 Zeilen auf den Monitor bringen kann. Zu diesen Unterschieden kommt es durch die verschiedenen Fernseh-Normen, wobei die deutsche "PAL"-Version des Amiga eben den Vorteil hat, mehrere Zeilen darstellen zu können, demgegenüber kann die amerikanische Version mit einem flimmerfreieren Bild aufwarten. In diesem Zusammenhang beachten Sie bitte, daß wir im folgenden von der PAL-Version ausgehen. Sollten Sie also noch eine Version der Workbench 1.1 benutzen, sollten Sie sich schleunigst eine neuere Version besorgen, da diese Version von maximal 400 Zeilen ausgeht und somit nicht die volle Auflösung unterstützt. Doch nun zu den einzelnen Grafikmodi:

(1.) LoRes:
Im LoRes-Modus (Low Resolution; Niedrige Auflösung) stehen uns auf dem Bildschirm 320 (x-Achse) mal 256 (x-Achse) Pixel mit je 32 Farbmöglichkeiten zur Verfügung. Die Beschränkung auf 32 Farben ergibt sich durch die begrenzte Anzahl von 32 Farbregistern.

(2.) HighRes:
Der HighRes-Modus (High Resolution; Hohe Auflösung) erlaubt uns eine Auflösung von 640 (x-Achse) mal 256 (y-Achse) Pixel, die jeweils eine von 16 Farben annehmen können. In diesem Modus kann der Amiga maximal vier Bitplanes anlegen, deshalb 16 (2 hoch 4) Farben.

(3.) Extra Halfbright:
Der Extra-Halfbright-Modus ist eine Sonderform des LoRes, wobei dieser Modus jedoch in der Lage ist, sechs Bitplanes anzulegen und somit 64 (2 hoch 6) verschiedene Farben auf den Bildschirm zu bringen. Dabei gibt es jedoch eine Einschränkung, denn wie bereits erwähnt, stehen dem Amiga nur 32 Farbregister zur Verfügung. Im Extra-Halfbright-Modus (wörtlich übersetzt: ("Besonderer-Halbhell-Modus") kann man selbst nur 32 Farben über die Farbregister aussuchen, die restlichen 32 Farben errechnet sich der Amiga aus den 32 zugeordneten. Dabei verringert er die Farbwerte so, daß eine in etwa halb so helle Farbe entsteht. So wird beispielsweise aus einem Rot ein Orange, aus einem Dunkelgrau eben ein Hellgrau.

(4.) Hold-And-Modify:
Der H.A.M.- Modus ist ebenfalls eine Besonderheit des LoRes und erlaubt es, alle 4096 Farben auf den Bildschirm zu bringen. Allerdings gibt es auch hier Einschränkungen, ebenfalls hervorgerufen durch die Farbregister.

Im Gegensatz zu den anderen Modi besteht nicht nur die Möglichkeit, den einzelnen Pixeln direkt über den Farbregistern eine bestimmte Farbe zuzuordnen, sondern man kann ihm eine Farbe relativ zum letzten Pixel geben. Konkret kann jeweils eines der drei Farbsignale Rot, Grün und Blau verändert, bzw. neu eingestellt werden. Um eine Farbveränderung über alle drei

Farbsignale zu erreichen, muß man sich also über eine Strecke von drei Pixels bewegen. Die ersten zwei Bitplanes sind dafür verantwortlich, ob der Pixel einen direkten Farbwert, oder einen relativen annehmen soll. Also ob das Rot, Grün oder Blau Signal verändert werden soll, oder ob die vier Bits der letzten vier Bitplanes dem Pixel über die Farbregister direkt einen Farbwert zuweisen.

(5.) Interlace:
Der Interlace - Modus läßt sich mit allen oben genannten Grafikmodi kombinieren. Er vergrößert die vertikale Auflösung (y-Achse) von 256 auf 512, allerdings verstärkt sich dabei das Flimmern des Bildschirms. Der Amiga teilt sich das vergrößerte Bild in zwei Hälften ein, die er abwechselnd um einen halben Pixel (vertikal) versetzt auf den Bildschirm bringt. Dabei enthält das erste Teilbild alle ungeraden Zeilennummern (1,3,5,...,511) und das zweite alle geraden(2,4,6,...,512). Das Flimmern entsteht, da jetzt nicht mehr 50, sondern nur noch 25 Bilder pro Sekunde auf dem Bildschirm erscheinen (Bei der amerikanischen Norm von 60 Hz sind es immerhin noch 30 Bilder pro Sekunde).

3. Speicherplatzbelegung einzelner Grafikmodi

Die Speicherplatzbelegung kann man wie folgt selbst für jeden Modi berechnen: Die Anzahl der Pixel, und somit auch der Bits einer Bitplane erhält man durch das Produkt der Pixel der x-Achse und der Pixel der y-Achse. Im LoRes (und nicht Interlace) also:

320 x 256 = 81920

Teilen wir durch 8, erhalten wir die Anzahl an Bytes, die eine Bitplane im Speicher belegt, in diesem Fall 10240 Bytes, also genau 10 KByte (1KByte = 1024Bit). Bei HighRes und Interlace sind es dann schon 40 KByte pro Bitplane. Nun ist die Anzahl der Bitplanes zwar auf sechs beschränkt, um jedoch noch einmal den Speicheraufwand von Grafiken zu verdeutlichen noch ein abschließendes Rechenbeispiel: Wäre es möglich bei höchster Auflösung alle 4069 Farben direkt addressiert auf den Bildschirm zu bringen, benötigte man zwölf Bitplanes (2 hoch 12 = 4096) und einen Speicherplatz von:

640 x 512 x 12 Bit = 480 KByte!

4. Die Playfields

Als Playfield (Spielfeld) wird beim Amiga das grafische Ausgabefenster bezeichnet. Es ist eine Arbeitsfläche, die sich vorzugsweise über den ganzen Bildschirm erstreckt und einen beliebigen Grafikmodus annehmen kann. Dabei muß das Playfield nicht unbedingt die gleiche Größe, wie das eigentliche Ausgabefenster haben, sondern darf auch größer sein.

Es wird dann jedoch immer nur ein Ausschnitt des Playfields gezeigt, der aber durch den Verschiebebefehl *Scroll* ruckfrei (SmoothScrolling) über das ganze Playfield verschoben werden kann. Eine weitere Besonderheit sind die Dual-Playfields, bei denen zwei Playfields sich so überlagern können, daß ein Play-

field im Vordergrund und eines im Hintergrund erscheint. Als Beispiel hierzu stellen Sie sich einen Flugsimulator vor, bei dem das Playfield im Vordergrund das Cockpit zeigt, während auf dem Playfield im Hintergrund alles das dargestellt wird, was sich außerhalb des Cockpits befindet.

5. Die Grafikhardware

Betreffend der Sonderchips des Amigas interessieren uns im Zusammenhang mit Grafik und ihrer Darstellung auf dem Bildschirm vor allem ***Agnus*** und ***Blitter***, wobei ***Agnus*** für die gesamte Videoausgabe, also das Umwandeln der Bitplanes in Video- bzw. RGB-Signale,verantwortlich ist. Für den Grafikprogrammierer ist jedoch der ***Blitter*** der wichtigere. Seine eigentliche Aufgabe ist es, Daten der Bitplanes zu kopieren, verschieben und zu verändern. Man kann mit ihm relativ einfach verschiedene Teile der Bitplanes, z.B. die Fenster, in andere Speicherbereiche kopieren, also verschieben. Des weiteren beinhaltet sein Befehlssatz auch Befehle zum Zeichnen von Linien zwischen zwei Punkten und zum Füllen von Flächen. Gerade beim Füllen von Flächen erkennt man die wahre Geschwindigkeit des Blitters. Obwohl Pixel für Pixel gesetzt wird, sieht es so aus, als ob auf dem Bildschirm eine Fläche einfach nur "eingeschaltet" und nicht erst aufgebaut wird.

Eine Besonderheit der Grafikhardware, die für die Programmierung von besonderer Bedeutung ist, ist die Tatsache, daß die Grafikchips nur auf das erste (je nach Versiobn halbe oder ganze) MByte zugreifen

können. Dieser Speicherbereich wird daher CHIP-Memory genannt. Wir werden sehen, daß es daher in vielen Fällen wichtig ist, daß bestimmte Daten in diesem Speicherbereich liegen. Das Betriebssystem stellt uns einige Prozeduren zur Verfügung, mit deren Hilfe man einen CHIP-Memory Bereich für eigene Zwecke reservieren kann. Damit Ihre Programme auf allen Amiga Versionen lauffähig sind, sollten Sie CHIP-RAM immer explizit allozieren. Sie sollten dies immer beachten, auch wenn Ihr Rechner nur 512 KByte hat (also nur CHIP-Memory).

6. Die Textausgabe

In der Textausgabe unterscheidet sich der Amiga durchaus von anderen Computersystemen. Während viele Computer eine getrennte Text-und Grafikausgabe haben, verfügt der Amiga gewissermaßen nur über die Grafikausgabe. Bei vielen anderen Computern wäre eine reine Grafikausgabe zu langsam, so daß bei der Textausgabe meistens nur die ASCII-Codes der Schriftzeichen im Speicher stehen und nicht deren grafischen Binärcodes. Der Amiga ist aber besonders auf dem Gebiet der Grafik einer der schnelleren Rechner, so daß diese Technik nicht nötig ist und der Text ebenfalls als Grafik ausgegeben wird. Dadurch lassen sich Schriftarten aber auch einfacher verändern und es sind auch Zeichensätze (die Fonts des Amiga) möglich, die vor allem nicht an eine bestimmte Größe gebunden sind. Als Beispiel hierzu ist das Programm Notepad aus der Workbench zu empfehlen. Mit ihm lassen sich verschiedene Zeichensätze laden, benutzen und auch ausdrucken.

Kapitel 2

========================
Fenster und Screens
========================

Im ersten Kapitel haben wir uns mit den durch die Hardware bedingten Aspekten der Videodarstellung beschäftigt. Für die Grafikprogrammierung ist es aber unabdingbar, sich auch mit der Art und Weise auseinanderzusetzten, wie die Systemsoftware die Möglichkeiten der Hardware ausnutzt und die Videoanzeige verwaltet. Die für den Benutzer sichtbare Anzeige mit den bekannten Fenstern, Menus, Gadgets etc., auf der wir später auch unsere Grafiken darstellen wollen, wird von einem Teil des Betriebssystems verwaltet, der ***Intuition*** genannt wird. Die Programmierung der Intuition ist ein sehr komplexes Thema, das allein schon ausreichen würde, um dieses Buch zu füllen. Wir werden uns deswegen hier auf eine kurze Einführung der wichtigsten Eigenschaften beschränken. Gegen Ende dieses Kapitels wird auch das Programm Display.h vorgestellt, das einige wichtige Routinen implementiert, die im Verlauf dieses Buches benötigt werden. Es befindet sich auch im "include" Verzeichnis der Begleitdiskette, aus welchem Sie es unbedingt in das gleichnamige Verzeichnis ihrer Arbeitsdiskette kopieren sollten, falls Sie mit den hier abgedruckten Programmen experimentieren wollen.

1. Was ist ein Screen?

Vieleicht haben Sie schon bei irgendeinem Programm gesehen, daß es möglich ist, zur gleichen Zeit mehrere Bildteile mit verschiedener Auflösung zu erzeugen, wobei jeder der Bildteile samt Inhalt hoch und runter bewegt werden kann. Ein solcher Bildteil wird vom Englischen her Screen (Bildschirm) genannt und steht an der untersten Stufe der Intuition-Anzeige. Ein Beispiel dafür ist der Workbench-Screen, der immer

dann erscheint, wenn Sie eine Diskette "gebootet" haben. Es können auch mehrere solcher Screens übereinander liegen. Man kann dann mit Hilfe des Depth-Gadgets (der kleine schwarzer Kasten in der rechten oberen Ecke) den momentan im Vordergrund liegenden Screen nach hinten befördern, wobei dann der darunterliegende zum Vorschein kommt. Auf einem Screen können Sie dann entweder direkt verschiedene Objekte darstellen (z.B. Text oder Grafik) oder ein bzw. mehrere Fenster aufmachen, die dann zur Darstellung gebraucht werden. Für uns liegt die besondere Bedeutung der Screens darin, daß man für eine Anzeige mit einer Auflösung, die nicht mit der der Workbench übereinstimmt, einen eigenen Screen aufmachen muß. Dabei wird auch die maximale Anzahl von Farben und ihre RGB-Zusammensetzung festgelegt. Auch die im ersten Kapitel angesprochenen Sondermodi: H.A.M. und Dual-Playfield können nur auf einem extra dafür erstellten Screen verwendet werden. Beim Öffnen eines Screens reserviert das Betriebssystem den für die Bitplanes notwendigen Speicher, in dem dann die Bilddaten geschrieben werden, und initialisiert die Farbregister. Wie Sie sehen, ist ein Screen im Grunde eine eigenständige, softwaremäßig verwaltete Videoanzeige. Die Leistung der Intuition besteht unter anderem darin, der Hardware mitzuteilen, aus welchem Screen sie die Bild- und Farbdaten lesen muß. Dazu wird einfach in dem Hardwaregister, der auf den aktuellen Bildschirmspeicher zeigt, die Adresse der Bitmap des entsprechenden Screens hineingeschrieben. Um mehrere Screens gleichzeitig untereinander darzustellen, wird erneut umgeschaltet, sobald der Strahl, der das Bild zeichnet, die Zeile erreicht hat, in der der neue Screen anfängt. Es ist leider nicht möglich, verschiedene Screens nebeneinander darzustellen, sodaß ein Screen immer ganz Links anfangen muß. Eine weitere Einschränkung ist dadurch gegeben, daß der untere Rand eines jeden Screens am unteren Bildschirmrand liegen

muß. Folglich muß die Summe aus der Screenhöhe in Zeilen und der Nummer der Zeile, in der die obere Grenze des Screens liegt, die maximale Zeilenanzahl bei der verwendeten Auflösung ergeben (512 bei Interlace, sonst 256.).

2. Öffnen eines Screens

Nachdem Sie nun mit den wichtigsten Eigenschaften eines Screens vertraut sind, können wir uns daran machen, einen eigenen Screen zu erzeugen. Zu diesem Zweck stellt Intution die Funktion ***OpenScreen*** zur Verfügung. Diese Routine verlangt als Eingabe einen Zeiger auf die ***NewScreen***-Struktur, in der sich alle Informationen über die gewünschten Eigenschaften des neuen Screens befinden. Dieser Datensatz sieht folgendermaßen aus:

Abb. 2.1 Die Screen-Struktur.

```
struct NewScreen {

SHORT   LeftEdge, TopEdge
        Width, Height,
        Depth;
UBYTE   DetailPen, BlockPen;
USHORT  ViewModes,
        Type;
struct  TextAttr  *Font;
UBYTE             *DefaultTitle;
struct  Gadgets   *Gadgets;
struct  Bitamp    *CustomBitMap;

};
```

Die Felder ***LeftEdge*** und ***TopEdge*** geben die Koordinaten der linken oberen Ecke des Screens an. Dabei muß ***LeftEdge*** immer 0 sein! ***Width*** und ***Height*** sind entsprechend die Breite und Höhe. Bei der Breite sollten Sie darauf achten, daß diese mit der Auflösung (siehe unten) übereinstimmt. ***Depth*** ist für die Anzahl der Bitebenen, also für die maximale Anzahl der Farben zuständig. Dabei ist die Farbenanzahl 2 hoch ***Depth***. Beachten Sie bei der Wahl der Tiefe die im ersten Kapitel beschrieben Einschränkungen, die sich aus der gewählten Auflösung ergeben. Die beiden Parameter: ***DetailPen*** und ***BlockPen*** bestimmen die Farbe der Schrift im Screenbalken und des Balkens selbst. Der Titel, der in dem Screenbalken erscheint, ist durch ****Default Title*** gegeben. ***Type*** gibt an, ob es sich um einen Workbench-Screen oder um einen besonderen, vom Benutzer erstellten Screen handelt und kann in der aktuellen Intuition - Version nur die Werte ***WBENCHSCREEN*** und ***CUSTOMSCREEN*** annehmen. Wenn Sie Ihren eigenen Screen aufmachen müssen Sie ***Type*** auf ***CUSTOMSCREEN*** setzen. Am wichtigsten ist das Feld ***ViewModes***, da es die Auflösung und die Sondermodi des Screens kontrolliert. Die folgende Tabelle wird Ihnen über die Werte, die die Auflösung bestimmen, Aufschluß geben:

Abb 2.2 Die Auflösung eines Screens

Gesetzte Flags	Resultierende Auflösung
NULL	320 x 256
HIRES	640 x 256
INTERELACE	320 x 512 (interlace)
HIRES ¦ INTERLACE	640 x 512 (interlace)

Zusätzlich gibt es noch die ***HAM- und DUALPF-Flags***, durch die Sie die H.A.M bzw Dual-Playfield Modi einschalten können. Ein H.A.M Screen mit der Auflösung 640 x 512 kann also durch den folgenden Wert von ***ViewModes*** erzeugt werden:

HIRES | INTERLACE | HAM

Die übrigen Felder der ***NewScreen***-Struktur haben für uns momentan keine Bedeutung und sollten mit **NULL** initialisiert werden. Wir werden auf sie in späteren Kapiteln zurückkommen.

Wenn Sie nun die ***OpenScreen***-Funktion aufrufen, werden Sie einen Zeiger auf die ***Screen***-Struktur von Intution bekommen. In dieser Struktur speichert das Betriebssystem die wichtigsten Daten eines Screens, wie z.B. Größe, Position, Koordinaten des Mauspfeils etc. Eine genaue Beschreibung der ***Screen***-Struktur finden Sie im Anhang A.

Zum Abschluß dieses Abschnittes folgt ein Beispielprogramm, das ein lowRes (320x256) Screen öffnet, einen Augenblick wartet und ihn wieder schließt.

Programm 2.1 Screen

```
#include "exec/types.h"
#include "intuition/intuition.h"


struct  IntuitionBase *IntuitionBase;

/*  NewScreen-Struktur zum Öffnen des Screens   */
struct  NewScreen NewScreen =
```

```
  {
  0,            /* x-Koordinate, muß 0 sein !     */
  0,            /* y-Koordinate.                  */
  320,          /* Breite des Screens.            */
  200,          /* Höhe des Screens.              */
  4,            /* Tiefe = 4 --> max 16 Farben.   */
  0,            /* Vordergrundfarbe.              */
  1,            /* Hintergrundfarbe.              */
  NULL,         /* "Normaler" lowres-Screen.      */
  CUSTOMSCREEN, /* Screentyp: Eigener Screen.     */
  NULL,         /* Standardfont benutzen.         */
  "Beispielscreen", /* Name des Screens.          */
  NULL,         /* Keine besonderen Gadgets.      */
  };

main ()
{
  struct  Screen *Screen;
  LONG     j;

/*  IntuitionLibrary öffnen, falls Fehler dann
                                      Programm beenden.*/

 IntuitionBase = (struct IntuitionBase *) OpenLibrary
("intuition.library",0);
if (IntuitionBase == NULL)
exit(FALSE);

/*  Den neuen Screen öffnen, falls Fehler, dann
                                    Programm beenden.   */
Screen  = (struct Screen *)
OpenScreen(&NewScreen);  if (Screen == NULL) exit
(FALSE);

  for (j = 0; j < 1000000; j++);          /* Abwarten */

  CloseScreen(Screen);             /* Screen schließen */
}
```

Beachten Sie, daß vor einem Zugriff auf die Strukturen und Prozeduren der Intution die entsprechende Library geöffnet werden muß. Es ist auch ratsam, immer wieder durch Vergleichen des Resultats mit ***NULL*** zu prüfen, ob beim Öffnen der Library bzw. später des Screens kein Fehler aufgetreten ist.

In der Intuition-Library gibt es neben ***OpenScreen*** und ***CloseScreen*** noch folgende Routinen zum Umgang mit den Screens:

MakeScreen
MoveScreen
ScreenToBack
ScreenToFront
ShowTitle

3. Die Fenster

Daß ein Fenster eines "dieser Rechtecke, die man auf dem Bildschirm schieben, verkleinern etc. kann, und in denen alles Mögliche erscheint" ist, sollte eigentlich jeder Amiga Benutzer wissen. Um aber die Fenster sinnvoll zur grafischen Darstellung verwenden zu können muß man sich auch die Art und Weise anschauen, wie sie von Intution verwaltet werden. Ein Fenster ist eigentlich ein selbständiger Terminal, in dem ein Programm samt Ein-/Ausgabe ablaufen kann. Es wird beim Öffnen einem Screen zugewiesen, und kann nur als sein Teil existieren. Es kann natürlich auch nicht aus ihm herausbewegt werden. Die Auflösung, RGB-Zusammensetzung der Farben und Anzeigemodi werden wie schon gesagt von dem Screen übernommen.

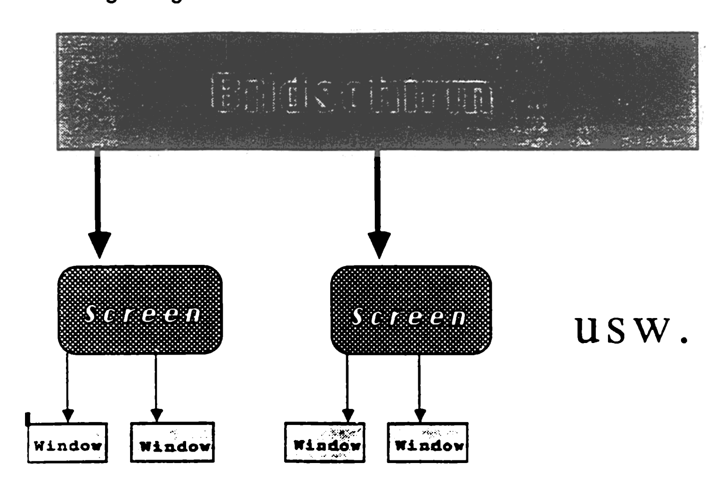


Bild 2.1. Die Intuition-Anzeige

Gewöhnlich hat ein Fenster auch keine eigene Bitmap, was z.B. daran zu erkennen ist, daß beim CLI-Fenster der Inhalt gelöscht wird, wenn das Fenster verkleinert wird. Andererseits geht aber der Inhalt des CLI-Fensters nicht verloren, wenn dieses in den Hintergrund geklickt wird. Unter welchen Umständen der Fensterinhalt beibehalten wird, hängt von dem sogenannten Refresh-Modus ab (engl. auffrischen), der beim Öffnen des Fensters eingestellt wird. Es stehen folgende Modi zur Verfügung:

(1) SimpleRefresh: In diesem Modus wird der Inhalt des Fensters nie zwischengespeichert, was zur Folge hat, daß beim Verkleinern, oder Verdecken durch ein anderes Fenster der Inhalt verloren geht.

(2) SmartRefresh: In diesem Modus wird nur der Inhalt eines Fensters, das durch ein anderes überdeckt wird, zwischengespeichert, und später wieder hergestellt.

(3) SuperBitMap: Ein solches Fenster besitzt eine eigene, vom Screen unabhängige Bitmap, so daß sein Inhalt immer beibehalten wird. Der Nachteil dieses Verfahrens ist, daß es sehr speicheraufwendig ist.

Neben den verschieden Refresh-Arten gibt es noch einige weitere interessante Eigenschaften, mit denen ein Fenster ausgestattet werden kann. Hier sind sie alle samt kurzer Beschreibung aufgelistet. Wir werden später auf einige besonders wichtige noch genauer eingehen.

(1) Die Gadgets

Dies sind Objekte, die in dem Fenster gezeichnet werden, und jedesmal wenn sich der Mauspfeil über ihnen befindet, ein Ereignis auslösen können. Intuition stellt folgende Systemgadgets zur Verfügung:

Close-Gadget: Ein Quadrat mit einem Punkt, der sich in der linken oberen Ecke des Fensters befindet. Sein Klicken soll das Programm zum Schließen des Fensters veranlassen (Das Fenster wird aber nicht automatisch geschlossen).

Sizing-Gadget: Befindet sich in der rechten unteren Ecke des Fensters und erlaubt dem Benutzer seine Größe zu verändern (Die Größenveränderung wird automatisch von Intution vorgenommen).

Drag-Gadget: Befindet sich in der Titelleiste und ermöglicht dem Benutzer, das Fenster zu verschieben, wobei die Verschiebung automatisch von Intuition durchgeführt wird.

Depth-Gadget: Die beiden Quadrate, die sich in der rechten oberen Ecke des Fensters befinden. Durch Anklicken eines der beiden Quadrate kann der Benutzer Intution dazu veranlassen, das Fenster in den Vordergrund, bzw. in den Hintergrund zu setzen.

Zusätzlich besteht auch noch die Möglichkeit eigene Gadgets (Custom-Gadgets) beliebigen Aussehens zu entwerfen, deren Handhabung jedoch eine sehr umfangreiche Beschreibung benötigen würde, so daß wir sie hier nur der Vollständigkeit halber erwähnen.

(2) Die Menus:

Jedem Fenster kann ein Pulldown-Menu System zugeordnet werden. Es erscheint in der Screenleiste beim Drücken der rechten Maustaste. Das Arbeiten mit sol-

chen Menüs ist ebenfalls ein sehr komplexes Thema, das allerdings mit Grafik nichts zu tun hat, und deshalb ebenfalls hier nicht genauer erläutert wird.

(3) Der IDCMP-Port: Dies ist eine Schnittstelle, über die Intuition Ihr Programm über das Auftreten verschiedener Ereignisse in einem Fenster informieren kann (siehe Abschnitt 5).

(4) GZZ-Fenster: Ein Fenster hat gewöhnlicherweise einen Rahmen und einen Titelbalken. In einem "normalen" Fenster ist die Koordinate (0,0), die linke obere Ecke des Rahmens (genauer des Titelbalkens), die als Bezugspunkt für alle Positionsangaben wie z.B. beim Zeichnen, dient. Beim GZZ-Fenster beziehen sich alle Koordinatenangaben auf die linke obere Ecke der innerhalb des Rahmens liegenden Fläche, so daß der Programmierer die Rahmenbreite nicht mehr zu berücksichtigen braucht.

(5) Borderless-Fenster: Ein solches Fenster hat, wie der Name schon sagt keinen Rahmen. Falls Sie keine Gadgets und keinen Titel angegeben haben, erscheint auch kein Titelbalken.

(6) Backdrop-Fenster: Dieser Fenstertyp bleibt immer im Hintergrund, d.h. auch wenn bei einem anderen Fenster das Depth-Gadget angeklickt wird, kann dieses Fenster nicht nach Vorne geholt werden.

4 Öffnen eines Fensters

Das Öffnen eines Fensters spielt sich nach dem gleichen Prinzip wie das Öffnen eines Screens ab. Es muß eine Struktur, die ***NewWindow***-Struktur, mit den Werten für das gewünschte Fenster intialisiert, und ihre Adresse an die Intuition Funktion ***OpenWindow*** übergeben werden. Als Ergebnis bekommt man, falls das Öffnen erfolgreich verlaufen ist, einen Zeiger auf die ***Window***-Struktur, in der ***Intuition*** alle relevanten Information über das Fenster speichert. Ist ein Fehler aufgetreten, so gibt diese Prozedur ***NULL*** zurück. Die ***NewWindow***-Struktur ist folgendermaßen definiert:

Abb 2.3 Die NewWindow-Struktur

```
struct  NewWindow {4}
SHORT   LeftEdge, TopEdge;
SHORT   Width, Height;
UBYTE   DetailPen, BlockPen;
ULONG   IDCMPFlags;
ULONG   Flags;;
struct  Gadget *FirstGadget;
struct  Image  *CheckMark;
UBYTE   *Title;
struct  Screen *Screen;
struct  BitMap *BitMap;
SHORT   MinWidth, MinHeight;
SHORT   MaxWidth, MaxHeight;
USHORT  Type;

};
```

Die ersten sechs Felder haben die gleiche Bedeutung wie bei der ***NewScreen***-Struktur: x,y-Koordinate, Breite, Höhe, und die Farbe der Titelschrift, bzw. des Titelbalkens des Fensters. Die Koordinaten beziehen sich auf die linke obere Ecke des Screens. Die Summe aus der x-Koordinate und Breite bzw. y-Koordinate und Höhe darf dann selbstverständlich nicht größer sein als die Breite bzw. die Höhe des Screens.

Die IDCMP-Flags dienen dazu, einen IDCMP-Port einzurichten und zu gestalten (siehe Abschnitt 5). ***NULL*** bedeutet hier gar keinen IDCMP-Port. Die im vorigen Abschnitt besprochenen besonderen Eigenschaften des neuen Fensters, können mittels des ***Flags***-Feldes durch die folgenden ***Window***-Flags eingestellt werden :

Abb 2.4 Die Window-Flags

Flag!	Eigenschaft!
WINDOWSIZING	Sizing-Gadget
WINDOWDEPTH	Depth-Gadget
WINDOWCLOSE	Close-Gadget
WINDOWDRAG	Drag-Gadget
SMART_REFRESH	Smart-Refresh Modus
SIMPLE_REFRESH	Simple-Refresh Modus
SUPER_BITMAP	Ein SuperBitMap-Fenster
GIMMEZEROZERO	Ein GZZ-Fenster
BACKDROP	Ein Backdrop-Fenster
BORDERLESS	Ein Fenster ohne Rahmen

Ein weiteres nützliches Flag ist das ACTIVATE-Flag, das darüber entscheidet, ob ein Fenster beim Öffnen aktiv wird oder nicht. Die Texteingabe und Überwachung der Mausaktivitäten können nur im aktiven Fenster erfolgen. Um nun z.B. ein mit den Close-und Drag Gadgets und ***SmartRefresh*** ausgestattetes Fenster zu öffnen, müssen sie folgende Flags setzen:

WINDOWDRAG ¦ WINDOWCLOSE ¦ SMART_REFRESH

In dem Feld ***Title*** müssen Sie die Adresse einer Zeichenkette, die dann im Titelbalken des Fensters erscheinen wird, übergeben. Die Adresse des Screens, auf dem das neue Fenster erscheinen soll, muß in ***Screen*** übergeben werden. Falls Sie hier ***NULL*** übergeben, wird das Fenster auf dem Workbench-Screen geöffnet. Je nachdem, ob Sie sich für einen eigenen, oder den Workbench-Screen entscheiden, setzen Sie ***Type*** auf ***CUSTOMSCREEN*** (eigener Screen) oder ***NULL*** (Workbench Screen).

Die Variablen ***MinWidth, MinHeight, MaxWidth,*** und ***MaxHeight*** bestimmen die minimalen bzw. maximalen Dimensionen des Fensters. Diese Felder sind nur für Fenster, die mit einem Drag-Gadget ausgestattet sind, wichtig. Sie bestimmen dann, inwieweit der Benutzer die Größe des Fensters verändern kann. Die restlichen Komponenten der ***NewWindow***-Struktur sind für uns im Moment nicht von Bedeutung und sollten mit Null initialisiert werden. Sie können Ihre Bedeutung dem Anhang A entnehmen.

Nach dieser Einführung kommt nun ein Beispiel, in dem ein einfaches Fenster ohne Gadgets und IDCMP-Port mit ***SmartRefresh*** auf dem Workbench-Screen geöffnet, und nach einer kurzen Pause wieder gechlossen wird.

Programm 2.2 Fenster.

```
#include "exec/types.h"
#include "intuition/intuition.h"


struct  IntuitionBase *IntuitionBase;


main ()

{
  struct  NewWindow NewWindow;
  struct  Window *Window;
  LONG     j;

/*  Intution-Library Öffnen, falls Fehler, dann
                                      Programm beenden. */

IntuitionBase = (struct IntuitionBase *)
OpenLibrary ("intuition.library",0);
  if (IntuitionBase == NULL) exit(FALSE);

/*  NewWindow-Struktur zum Öffnen des Fensters
                                        initialisieren  */

  NewWindow.LeftEdge = 1;  /*  x-Koordinate.        */
  NewWindow.TopEdge = 1;   /*  y-Koordinate.        */
  NewWindow.Width = 300;   /*  Breite des Fensters.*/
  NewWindow.Height = 150;  /*  Höhe des Fensters.   */
  NewWindow.BlockPen = 1;  /*  Vordergrundfarbe.    */
  NewWindow.DetailPen = 0; /*  Hintergrundfarbe.    */
                           /*  Name des Fensters.   */
  NewWindow.Title = "BeispielFenster";
               /*  Keine Gadgets; "Smart-Refresh".  */

  NewWindow.Flags = SMART_REFRESH | ACTIVATE;
```

```
  NewWindow.IDCMPFlags = NULL;
                    /*  Kein Nachrichtenport (IDCMP).    */

  NewWindow.Type = WBENCHSCREEN;
                    /*  Fenstertyp: Workbenchfenster.    */

  NewWindow.FirstGadget = NULL;
                    /*  Keine Gadgets.                   */

  NewWindow.CheckMark = NULL;
                    /*  Standard "Checkmark".            */

  NewWindow.Screen = NULL;
                    /*  Auf dem Workbench-Screen.        */

  NewWindow.BitMap = NULL;
                    /*  Keine eigene Bitmap.             */

  NewWindow.MinWidth = 1;
                    /*  Minimale Breite = 1.             */

  NewWindow.MinHeight = 1;
                    /*  Minimale Höhe = 1.               */

  NewWindow.MaxWidth = 300;
                    /*  Maximale Breite unwichtig.       */

  NewWindow.MaxHeight = 150;
                    /*  Maximale Höhe unwichtig.         */

 /*  Das neue Fenster öffnen, falls Fehler, dann
                                  Programm beenden.   */

Window  = (struct Window *)
OpenWindow(&NewWindow);if (Window==NULL) exit(FALSE);
  for (j = 0; j < 1000000; j++);         /* Abwarten */
  CloseWindow(Window);           /* Fenster schließen */
}
```

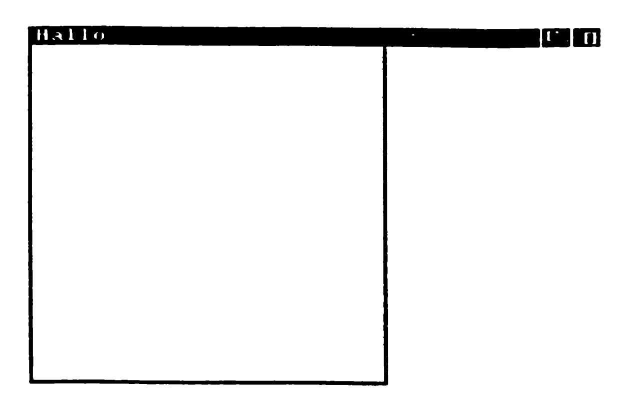


Bild 2.2 - Die Ausgabe des Programms Fenster

Zum Abschluß noch eine kurze Aufzählung der Intuition-Prozeduren zum Arbeiten mit Fenstern.

ActivateWindow
CloseWindow
MoveWindow
OpenWindow
SetWindowTitles
SizeWindow
WindowLimits
WindowToBack
WindowToFront

5 Der IDCMP-Port

Eine der Besonderheiten des Amiga ist die Tatsache, daß alle Eingaben, sowohl die, die über die Tastatur, als auch die, die durch die Maus erfolgen, immer nur fensterbezogen sind. So wird z.B. ein Text nur in dem Fenster eingegeben, in dem zuletzt die Maus geklickt wurde. Der IDCMP-Port ist die Schnittstelle, die dafür sorgt, daß Ihr Programm alles, was in einem Fenster passiert, überwachen kann. Durch Setzen der entsprechenden Flags können Sie Intuition dazu veranlassen, Ihnen jedesmal eine Nachricht zu schicken, wenn der Benutzer in einem Fenster z.B. die Maus bewegt, ein Menü auswählt,oder eine Taste drückt. Das Setzen dieser Flags kann entweder direkt beim Öffnen des Fensters durch entsprechendes Initialisieren des ***IDCMPFlags***-Feldes der ***NewWindow***-Struktur erfolgen, oder später mit Hilfe der Intuition Prozedur ***Modify-IDCMP***. Falls Sie also immer dann, wenn die Maus bewegt wird oder wenn das Close-Gadget geklickt wird, informiert werden wollen, müssen Sie diesem Feld den Wert:

MOUSEMOVE ¦ CLOSEWINDOW

zuweisen, oder ***ModifyIDCMP*** folgendermaßen aufrufen:

ModifyIDCMP(Window, MOUSEMOVE ¦ CLOSEWINDOW);

Window muß selbstverständlich ein Zeiger auf die ***Window***-Struktur Ihres Fensters sein!

Hier die wichtigsten *IDCMP-Flags* und die zugehörigen Ereignisse:

MOUSEBUTTONS
Drücken/Loslassen einer Maustaste. Je nach dem ob das *MBTRAP* Window-Flag gesetzt ist, wird dabei die rechte Taste mit berücksichtigt oder nicht.

MOUSEMOVE
Veränderung der Mausposition.

CLOSEWINDOW
Ancklicken des Close-Gadgets.

NEWSIZE
Veränderung der Fenstergröße.

ACTIVEWINDOW/INACTIVEWINDOW
Falls Sie dieses Flag setzen, werden Sie eine Nachricht erhalten, wenn das Fenster durch Klicken der Maus in seinem innerem aktiviert, bzw. durch Klicken in einem anderem Fenster inaktiviert wird.

VANILLAKEY
Die gedrückte Taste.

DISKINSERTED/DISKRENOVED
Sie bekommen eine Nachricht, falls eine Diskette eingelegt bzw. aus dem Laufwerk herausgenommen wurde.

Nachdem Sie nun einen IDCMP-Port entsprechend Ihrer Bedürfnisse eingerichtet haben, können Sie diesen jederzeit während des Programmablaufes abfragen. Normalerweise wird dabei zuerst die Exec-Prozedur ***WaitPort*** aufgerufen, die dafür sorgt, daß Ihr Programm solange angehalten wird, bis irgendeine Nachricht an das Fenster gekommen ist. Hiernach kann mit ***GetMsg*** ihre Adresse ermittelt werden. Beide Prozeduren brauchen als

Eingabe lediglich das ***UserPort***-Feld der ***Window***-Struktur. Bei dem Zeiger, den ***GetMsg*** zurück gibt, handelt es sich um die Adresse der ***IntuiMessage***-Struktur, in der die Nachricht gespeichert ist. Diese ist wie folgt aufgebaut:

Abb 2.5 Die IntuiMessage-Struktur

```
struct IntuiMessage {

struct   Message ExecMessage;
ULONG    Class;
USHORT   Code;
USHORT   Qualifier;
APTR     IAddress;
SHORT    MouseX, MouseY;
ULONG    Seconds, Micros;
struct   Window *IDCMPWindow;
struct   IntuiMesage *SpecialLink;

};
```

Die Art der Nachricht kann dem Feld ***Class*** entnommen werden. Eventuelle zusätzliche Informationen, wie z.B. der ASCII-Code des Zeichens, der der gedrückten Taste entspricht, stehen in der ***Code***-Variable. In ***Mousex und Mousey*** finden Sie immer die aktuellen Koordinaten des Mauszeigers. Falls seit der letzten Abfrage des Ports zwei Ereignisse aufgetreten sind, die entweder vom gleichen Typ sind oder beide das ***Code***-Feld für sich beanspruchen, wird das zweite Ereignis als eine weitere Nachricht eingetragen, die in eine Warteschleife eingefügt wird. Nachdem die Nachricht empfangen und untersucht wurde, muß durch Aufruf der ***ReplyMsg***-Routine dies dem System mitgeteilt werden. Nun wird die alte ***IntuiMessage***-Struktur vergessen und Sie können auf die eben beschriebene Wei-

se die nächste Nachricht holen. Um Ihnen das ganze auf den ersten Blick vielleicht ein wenig verwirrende Verfahren zu veranschaulichen, haben wir hier wieder ein Beispielprogramm abgedruckt.

Es öffnet ein Fenster mit einem IDCMP-Port und wartet vor dem Schließen bis das Close-Gadget angeklickt wird. Die *while*-Schleife wäre an dieser Stelle eigentlich nicht notwendig, da dies das einzige Ereignis ist, daß von Intution in diesem Fall registriert wird (es ist nur das *CLOSEWINDOW*-Flag gesetzt !), so daß das Programm allein durch *WaitPort* angehalten wird.

Programm 2.3 IDCMP

```
#include "exec/types.h"
#include "intuition/intuition.h"


struct  IntuitionBase *IntuitionBase;

main ()

{
  struct  NewWindow NewWindow;
  struct  Window *Window;
  struct  IntuiMessage *IntuiMessage;
  LONG     j;
  ULONG    Type;
 /*  Intution-Library öffnen, falls Fehler,dann
                                      Programm beenden.*/
  IntuitionBase =(struct IntuitionBase *) OpenLibrary
("intuition.library",0);
  if (IntuitionBase == NULL) exit(FALSE);
```

```
/*  NewWindow-Struktur zum Öffnen des Fensters
                                      initialisieren  */
  NewWindow.LeftEdge = 1; /*  x-Koordinate.          */
  NewWindow.TopEdge = 1;  /*  y-Koordinate.          */
  NewWindow.Width = 300;  /*  Breite des Fensters.*/
  NewWindow.Height = 150; /*  Höhe des Fensters.     */
  NewWindow.BlockPen = 1; /*  Vordergrundfarbe.      */
  NewWindow.DetailPen = 0;/*  Hintergrundfarbe.      */
                          /*  Name des Fensters.     */
  NewWindow.Title = "BeispielFenster";
              /*  Alle Gadgets, "Smart-Refresh".     */

  NewWindow.Flags = SMART_REFRESH | ACTIVATE |
WINDOWCLOSE | WINDOWDRAG |WINDOWDEPTH | WINDOWSIZING;
             /*  Close Gadget Message schicken.      */

  NewWindow.IDCMPFlags = CLOSEWINDOW;
  NewWindow.Type = WBENCHSCREEN;
             /*  Fenstertyp: Workbench Fenster.      */

  NewWindow.FirstGadget = NULL;
             /*  Keine Gadgets.                      */

  NewWindow.CheckMark = NULL;
             /*  Standard "Checkmark".               */

  NewWindow.Screen = NULL;
             /*  Auf dem Workbench-Screen.           */

  NewWindow.BitMap = NULL;
             /*  Keine eigene Bitmap.                */

  NewWindow.MinWidth = 1;
             /*  Minimale Breite = 1.                */

  NewWindow.MinHeight = 1;
             /*  Minimale Höhe   = 1.                */
```

```
  NewWindow.MaxWidth = 300;
              /*  Maximale Breite unwichtig.       */

/* Das neue Fenster öffnen, falls Fehler, dann
                                    Programm beenden.  */
Window  = (struct Window *)OpenWindow(&NewWindow);
  if (Window == NULL) exit(FALSE);

  Type = 0;
  /*  Auf Nachricht warten bis Close-
                                    Gadget angeclickt.  */
  while (Type < CLOSEWINDOW)
  {
   /*  Auf Nachricht warten  */
   WaitPort(Window->UserPort);

   /*  Art der Nachricht merken.  */
   IntuiMessage = GetMsg(Window->UserPort);
   Type = IntuiMessage->Class;
  }

  /*  Fenster schließen   */
  CloseWindow(Window);

}
```

6 Das Display Programm

Wie Sie sehen, ist der Gebrauch der Fenster und Screens mit einigem Aufwand verbunden. Um uns in Zukunft diese Arbeit zu ersparen, wird nun ein Programm vorgestellt, in dem einige nützliche Prozeduren definiert werden. Wir werden dann später ohne weiteren Kommentar auf diese Routinen zugreifen.

Das Programm, das wir Display.h genannt haben, beinhaltet folgende Prozeduren:

OpenIntui ()
Diese Prozedur öffnet die Intution-Library.

OpenGfx ()
Diese Prozedur öffnet die Graphics-Library

MakeScr (x,y,w,h,Name,d,flags,font,BMap)
Öffnet einen neuen Screen, wobei die Parameter den wichtigsten Feldern der ***NewScreen***-Struktur entsprechen. Die Variablen ***x,y,w,h*** bestimmen die Koordinaten der linken oberen Ecke, die Breite und die Höhe. Die Tiefe wird in ***d*** angegeben. ***flags*** entspricht dem ***ViewModes***-Feld von ***NewScreen***, dient also zur Auswahl der Auflösung und der Sondermodi. In ***font*** bzw. ***BMap*** können Sie schließlich einen eigenen Font, bzw. ***BitMap*** für diesen Screen angeben. Normalerweise sollten diese Parameter auf NULL gesetzt werden.

MakeWindow (x,y,w,h,mw,mh,Name,flags,Idcmp,screen)
öffnet ein Fenster wobei die übergebenen Parameter die wichtigsten Felder der ***NewWindow***-Struktur bestimmen. Die Variablen ***x,y,w,h,mw,mh*** stehen in dieser Reihenfolge für die Koordinaten der linken oberen

Ecke, Breite, Höhe und die maximalen Abmessungen des Fensters. ***Name*** ist eine Zeichenkette, die den Titel des Fensters beinhaltet. In ***flags*** bzw. ***Idcmp*** können Sie die gewünschten ***Window*** bzw.***IDCMP***-Flags übergeben. Der letzte Parameter, ***screen***, ist ein Zeiger auf den Screen, auf dem das Fenster geöffnet wird. ***NULL*** bedeutet hier, den Workbench-Screen

WaitEvent (wind,code)
Diese Prozedur wartet, bis an das Fenster, deren Adresse in ***wind*** übergeben wurde, eine IDCMP-Nachricht kommt und gibt den Inhalt des ***Class***-Feldes der ***IntuiMessage***-Struktur (also die Art der Nachricht!) zurück. In ***code*** wird dabei der Inhalt des ***Code***-Feldes von ***IntuiMessage*** geschrieben.

GetMouse(wind,x,y)
ermittelt die aktuelle Position des Mauszeigers in dem durch ***wind*** bestimmten Fenster.

```
#include "exec/types.h"
#include "intuition/intuition.h"

struct  IntuitionBase *IntuitionBase;
struct  GfxBase *GfxBase;

/*  Diese Prozedur öffnet die Intution-Library  */
VOID OpenIntui ()
{
  /*  Intution-Library öffnen, falls Fehler dann
                                     Programm beenden.*/
  IntuitionBase = (struct IntuitionBase *)
                     OpenLibrary("intuition.library",0);
  if (IntuitionBase == NULL) exit(FALSE);
}
```

```
/*  Diese Prozedur öffnet die Graphics-Library  */
VOID OpenGfx ()
{
  GfxBase = (struct GfxBase *) OpenLibrary("graphics.
                                          library",0);
  if (IntuitionBase == NULL) exit(FALSE);
}


/*  Diese Prozedur öffnet einen neuen Screen.  */
MakeScr (x, y, w, h, Name, d, flags, font, BMap)
APTR    Name, BMap, font;
SHORT   x, y, w, h, d;
ULONG   flags;

{
  struct  NewScreen NewScreen;

  /*  NewScreen-Struktur initialisieren.  */
  NewScreen.LeftEdge      = x;
  NewScreen.TopEdge       = y;
  NewScreen.Width         = w;
  NewScreen.Height        = h;
  NewScreen.Depth         = d;
  NewScreen.DetailPen     = 0;
  NewScreen.BlockPen      = 1;
  NewScreen.ViewModes     = flags;
  NewScreen.Type          = CUSTOMSCREEN;
  NewScreen.Font          = font;
  NewScreen.Title         = Name;
  NewScreen.Gadgets       = NULL;
  NewScreen.CustomBitMap  = NULL;

  /*  Den neuen Screen öffnen.  */
  return (OpenScreen(&NewScreen));
}
```

```
/*  Diese Prozedur öffnet ein neues Fenster   */
MakeWindow (x, y, w, h, mw, mh, Name, flags, Idcmp,
                                              screen)
struct  Screen *screen;
APTR    Name;
SHORT   x, y, w, h, mw, mh;
ULONG   flags, Idcmp;
{
  struct  NewWindow NewWindow;
  ULONG SType;
   /*  Wokbenchfenster oder nicht ?  */
  SType = CUSTOMSCREEN;
  if(screen == NULL)
   SType = WBENCHSCREEN;

  OpenIntui();      /*  Intuition Library öffnen   */

  /*  NewWindow-Struktur initialisieren   */
  NewWindow.LeftEdge      = x;
  NewWindow.TopEdge       = y;
  NewWindow.Width         = w;
  NewWindow.Height        = h;
  NewWindow.BlockPen      = 1;
  NewWindow.DetailPen     = 0;
  NewWindow.Title         = Name;
  NewWindow.Flags         = flags;
  NewWindow.IDCMPFlags    = Idcmp;
  NewWindow.Type          = SType;
  NewWindow.FirstGadget   = NULL;
  NewWindow.CheckMark     = NULL;
  NewWindow.Screen        = screen;
  NewWindow.BitMap        = NULL;
  NewWindow.MinWidth      = 1;
  NewWindow.MinHeight     = 1;
  NewWindow.MaxWidth      = mw;
  NewWindow.MaxHeight     = mh;
  return (OpenWindow(&NewWindow));
}
```

```
/* Diese Prozedur wartet auf eine Intution Nachricht,
                                    holt die Art des    */
/* Ereignisses ("Class") und gegebenfalls die
                                gelesene Größe ("Code").*/
WaitEvent (wind, code)

struct Window *wind;
USHORT *code;

{
  struct IntuiMessage *IntuiMessage;

  /*  Auf Nachricht warten  */
  Wait(1 << wind->UserPort->mp_SigBit);

  /*  Nachricht holen.  */
  IntuiMessage = GetMsg(wind->UserPort);

  /*  Inhalt merken und Art der Nachricht
                                          zurückgeben.  */
  *code = IntuiMessage->Code;
  return (IntuiMessage->Class);
}
/*  Diese Prozedur holt die aktuellen Koordinaten
                            des Mauszeigers im Fenster. */
GetMouse(wind,x,y)

struct Window *wind;
SHORT *x, *y;
{
  *x = wind->MouseX;
  *y = wind->MouseY;

  /*  Ist dies ein GZZ-Fenster ?  */
   { *x = wind->GZZMouseX;
     *y = wind->GZZMouseY;
   };
}
```

Als Beispiel für die Anwendung der soeben definierten Routinen folgt ein Programm, das ein Fenster auf einem neuen Screen öffnet, auf Eingabe von "E" wartet und es wieder schließt.

Programm 2.5 DisplayDemo

```
#include "Display.h"

main ()
{
  struct Screen *Screen;
  struct Window *Window;
  ULONG  Class;
  short  Code;

  OpenIntui ();

  /*  Einen low-res Screen öffnen.  */
  Screen = MakeScr(0,0,320,200,"Lowres",2,NULL,
                                           NULL,NULL);
  if (Screen == NULL) exit(FALSE);

  /*  Fenster auf dem neuen Screen öffnen.  */
  Window = MakeWindow (0,0,200,200,200,200,"Hallo",
                  SMART_REFRESH, VANILLAKEY,Screen);
  if(Window == NULL)

  /*  Warten bis "E" gedrückt wird  */
  do{
      /*  Auf Nachricht warten  */
      Class = WaitEvent(Window,&Code);
    } while (Code != 69);

  /*  Fenster und Screen schließen  */
  CloseWindow(Window);
  CloseScreen(Screen);
}
```

Kapitel 3

Die Zeichenroutinen der Intuition

Obwohl sich die Mehrheit der Grafikprozeduren logischerweise in der ***Graphics***-Library befindet, bietet auch Intuition mit ***DrawImage*** und ***DrawBorder*** zwei sehr nützliche Routinen. Sie werden vom System zur Darstellung der Fenstergrenzen und der Gadgets benötigt und ermöglichen das Zeichnen von beliebigen Bildern und Umrandungen. Sie werden in diesem Kapitel im Zusammenhang mit den Images auch mit Routinen zur Allozierung und Deallozierung von Speicherbereichen vertraut gemacht werden, die wir später noch sehr oft benötigen werden. Besonderes wichtig ist dabei, daß Sie sich von Anfang an angewöhnen, nicht mehr benötigten Speicher auch wieder freizugeben, auch wenn es sich um scheinbar kleine Mengen handelt.

1 Zeichnen von Umrandungen: DrawBorder

Zur Umrandung diverser Objekte bietet Intuition die ***DrawBorder***-Prozedur. Diese benötigt eine mit entsprechenden Parametern initialisierte ***Border***-Struktur, in der unter anderem die Koordinaten der Eckpunkte enthalten sind. Nachdem Sie eine Umrandung(Border) durch Erstellen eines solchen Datensatzes erzeugt haben, können Sie diese mehrmals an verschiedenen Stellen des Bildschirms ausgeben. Auf diese Weise lassen sich mit einfachen Mitteln relativ komplexe Gebilde auf dem Bildschirm darstellen. Obwohl wir hier von Umrandungen sprechen, können nicht nur geschlossene Linienzüge gezeichnet werden. Durch die Möglichkeit, mehrere solcher Borders miteinander zu verketten, können auch nicht zusammehängende Figuren definiert und dargestellt werden. Der erste Schritt in der Definition einer Umrandung ist die Festlegung der Eckkoordinaten. Diese können als Zahlen in einem

SHORT-Arrray gespeichert werden. Ein Dreieck, dessen Spitze bei (10,0) liegt, kann also wie folgt definiert werden:

```
SHORT Dreieck [] = {10,0,
                    20,10,
                    0,10,
                    10,0};
```

Als nächstes müssen Sie eine ***Border***-Struktur folgender Form erzeugen:

Abb 3.1 Die Border-Struktur.

```
struct Border {

SHORT  LeftEdge, TopEdge;
SHORT  FrontPen, BackPen;
SHORT  Count;
SHORT  *XY;
struct Border *NextBorder;

}
```

In ***LeftEdge*** und ***TopEdge*** müssen Sie den sogenannten Offset eintragen, das heißt die x- und y-Entfernung beim Zeichnen mit ***DrawBorder*** angegeben wurde. Haben Sie also beim Zeichnen als Koordinaten z.B. (100,100) und als Offset (10,10) angegeben, dann wird der erste Punkt des in Abbildung 3.1 definierten Dreiecks bei

x = 100 + 10 + 10
y = 100 + 10

also (120,110) gezeichnet.

FrontPen bestimmt die Farbe, die zum Zeichnen benutzt wird, falls ***DrawMode*** den Wert ***JAM1*** hat.(***BackPen*** hat in der aktuellen Intuition Version keine Bedeutung). Im Falle des anderen zulässigen ***DrawMode***-Wertes: ***XOR***, wird beim Zeichnen die Farbe des Hintergrundes invertiert. Für die Form Umrandung sind die Variablen ***XY*** und ***Count*** verantwortlich. In der ersten geben Sie die Adresse des Feldes an, in dem die Koordinaten der Eckpunkte enthalten sind, während die zweite die Anzahl der Ecken bestimmt, die tatsächlich gezeichnet werden. Um von unserem Dreieck nur die zwei ersten Seiten zu zeichnen, müßte man ***Count*** gleich 3 setzen (Anfangspunkt, erste Ecke, zweite Ecke). Das letzte Feld der Border-Struktur, ***NextBorder***, ist dazu da, mehrere solcher Figuren zu verketten, indem dort die Adresse einer weiteren solchen Struktur eingetragen wird. Eine solche Kette kann dann mit einem einzigen ***DrawBorder***-Befehl gezeichnet werden. Der Aufruf dieser Prozedur sieht immer aus, wie folgt:

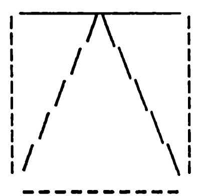
DrawBorder(RPort,Border,x,y)

In ***Border*** wird dann die Adresse der initialisierten ***Border***-Struktur, in ***x,y*** die Position, an der diese auszugeben ist, übergeben. ***RPort*** bezeichnet die Adresse eines Rastports. Was dieser nun genauer ist, erklären wir später. Um die Umrandung in einem Screen darzustellen, geben Sie hier die Adresse des ***RastPort***-Feldes der ***Screen***-Struktur an.

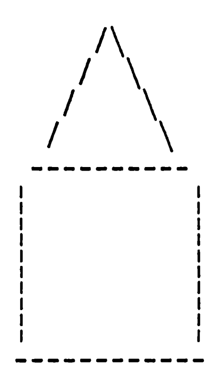
In dem folgenden abgedruckten Programm Border haben wir versucht, Ihnen vor allem die nicht ganz so triviale Anwendug der Offsets und der Verkettung von Umrandungen zu verdeutlichen. Dazu werden zwei zunächst nicht verkettete Borders definiert:

(1) ***Rect*** - Quadratisch
(2) ***Tria*** - Dreieckig

Sie werden dann in einer ***for***-Schleife untereinander mit zwei ***DrawBorder*** Aufrufen gezeichnet. Hiernach wird die Adresse von Tria in das ***NextBorder*** Feld von ***Rect*** eingetragen, und Rect wird wieder in einer for-Schleife gezeichnet. Da die beiden aber nun verkettet sind, wird ja das Dreieck mitgezeichnet und die resultierende Figur sieht in etwa so aus:



Zum Abschluß wird dann noch der y-Offset (***TopEdge***) von Tria auf -25 gesetzt, so daß das Dreieck nun über dem Quadrat, also wie folgt ausgegeben wird:



Programm 3.1 Border

```
#include "Display.h"
#include "intuition/intuition.h"
#include "exec/types.h"

struct Window *Window;

SHORT  Corners1 [] = {0,  0,        /*  Ein Quadrat   */
                      50, 0,
                      50, 25,
                      0,  25,
                      0,  0};

SHORT  Corners2 [] = {25, 0,        /*  Ein Dreieck   */
                      50, 25,
                      0,  25,
                      25, 0};

struct  Border Rect = {0, 0,
                       1, 0,
                       JAM1,
                       5,
                       &Corners1,
                       NULL};

struct  Border Tria = {0, 0,
                       1, 0,
                       JAM1,
                       4,
                       &Corners2,
                       NULL};
main ()
{
 SHORT  i;
 USHORT code;
 ULONG  Class;
```

```
/*  Ein Fenster auf dem Workbench Screen öffnen  */
Window = (struct Window *)MakeWindow(0,0,640,250,
                               640,250,"Border-Beispiel",
   WINDOWCLOSE | SMART_REFRESH, CLOSEWINDOW, NULL);

if(Window == NULL)      /*  Fehler beim Öffnen ?  */
 exit(FALSE);

for(i = 5; i <= 600; i = i + 55)
           /*  Beide Borders getrennt zeichnen.  */
{
  DrawBorder(Window->RPort,&Rect,i,20);
                        /*  Das Quadrat Zeichnen.  */
  DrawBorder(Window->RPort,&Tria,i,70);
                        /*  Das Dreieck Zeichnen.  */
}
/*  Zweite Umrandung an die erste "anhängen".  */
Rect.NextBorder = &Tria;

/*  Und beide 10 mal gleichzeitig zeichnen.  */
for(i = 5; i <= 600; i = i + 55)
{
  DrawBorder(Window->RPort,&Rect,i,130);
}

/*  Offset des Dreiecks verändern  */
Tria.TopEdge = -25;
/*  Und beide Borders wieder 10 mal gleichzeitig
                                     zeichnen.  */
for(i = 5; i <= 600; i = i + 55)
{
  DrawBorder(Window->RPort,&Rect,i,210);
}

Class = WaitEvent(Window,&code);
                  /*  Auf Close-Gadget warten.  */
CloseWindow(Window);   /*  Fenster schließen.  */
}
```

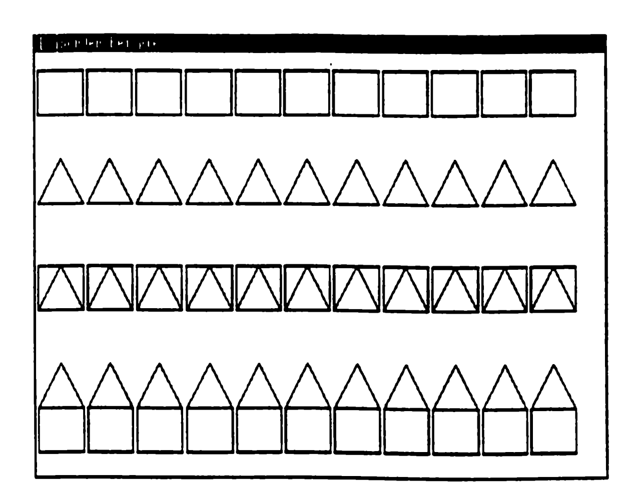


Bild 3.1 - Die Ausgabe des Programms Border.

2. Die Speicherallozierung

Bevor wir uns mit den Images (Bilder) befassen können, ist ein kleiner Exkurs in die Speicherverwaltung fällig. Wir müssen nämlich, um die Bilddaten für die Grafikchips zugängig zu machen, sicherstellen, daß sie sich im CHIP-Ram befinden. Tut man es nicht, so stürzt das Programm auf so manchem Amiga prompt ab.

Als erstes muß der für die Daten notwendige Speicherplatz mit Hilfe der Exec-Prozedur ***AllocMem*** reserviert (alloziert) werden. Dazu müssen die Größe des benötigten Speichers in Bytes und die Anforderungen, die an diesen Speicher gestellt werden, übergeben werden. Die letzteren werden durch ein Set folgender Flags bestimmt:

MEMF_CHIP: Es wird CHIP-Ram benötigt.

MEMF_CLEAR: Der Speicher soll beim Allozieren mit Nullen initialisiert werden.

MEMF_LARGEST: Es wird ein zusammenhängender Speicherblock benötigt.

Als Ergebnis liefert dann ***AllocMem*** einen Zeiger auf den reservierten Speicher. Um z.B. Speicher für Bildarten, die sich in der Variable Data befinden zu allozieren müssen Sie so vorgehen:

DataPtr=AllocMem(sizeof(Data),MEMF_CHIP¦MEMF_CLEAR);

Dabei wird die benötigte Speichermenge mit Hilfe der ***sizeof***-Funktion ermittelt.

Exec stellt auch eine Routine zum Kopieren von Speicherinhalten zur Verfügung. Sie heißt ***CopyMem*** und benötigt folgende Eingaben:

source: Die Adresse von der kopiert wird.
dest: Die Adresse zu der kopiert werden soll.
size: Die Größe des zu kopierenden Bereiches in Bytes.

Die Daten vom vorigen Beispiel können also so kopiert werden:

CopyMem(&Data,DataPtr,sizeof(Data));

Es werden auf diese Weise die Adresse und Größe des Speichers als Eingabe gebraucht. Also z.B.:

FreeMem(DataPtr,sizeof(Data));

Dabei ist Vorsicht geboten, denn ein Versuch, einen

nicht allozierten Speicher zu deallozieren, endet oft mit Absturz.

3 Die Images

Die Images (engl. Bilder) gestatten es Ihnen, das Aussehen von Objekten Pixel für Pixel zu bestimmen. Analog wie bei den Borders können sie dann mehrmals, auch verkettet, gezeichnet werden. Die Bilddaten werden in einer Folge von ***USHORT***-Zahlen (Prozessorwörter) gespeichert, deren Bitzusammensetzung der Pixelzusammensetzung des Bildes entspricht. Die Farbe wird hierbei nicht mehr einheitlich für das ganze Objekt angegeben, sondern wie bei der Videoanzeige durch überlappende Bitplanes bestimmt. Die Anzahl der Bitplanes eines Images kann zwischen 1 und der Tiefe des Screens, auf dem dieser ausgegeben wird, liegen (bzw. zu dem das Fenster, in dem dieser ausgegeben wird, gehört). Eintachheitshalber wollen wir uns zuerst mit einem Image der Tiefe 1, das die Form eines Kreuzes hat, beschäftigen. Zuerst müssen wir uns das Bild so wie es spät dargestellt wird, aufzeichnen, also mit Hilfe von gesetzten oder nicht gesetzten Punkten.

Abb 3.2 Ein einfaches Image

```
0000111111110000
0000111111110000
1111111111111111
1111111111111111
0000111111110000
0000111111110000
```

Durch die Darstellung in Form von Nullen und Einsen haben wir es geschafft, das Bild in eine Folge von 16 Bit Dualzahlen umzuwandeln. Diese können nun in Dezimal bzw. Hexadezimalsystem umgerechnet werden und entsprechen von der Länge jeweils einem Prozessorwort (***WORD***). Nach dieser Umwandlung könnte man nun unser Kreuz wie folgt als hexadezimale Bilddaten Speichern:

```
USHORT Bild [] = {0x0ff0,
                  0x0ff0,
                  0xffff,
                  0xffff,
                  0x0ff0,
                  0x0ff0};
```

Wie Sie sehen, entspricht bei der Umrechnung eine Hexadezimalziffer einer vier Bit langen Dualzahl.

In diesem Beispiel wurde das Bild so gewählt, daß es die Breite von 16 Punkten hat. Falls die Breite geringer ist, wird trotzdem jede Zeile durch eine 16-Bit Zahl bestimmt. Die nicht benutzten Bits müssen dann halt Null sein. Für Bilder, die breiter als die 16 Punkte sind, werden zur Definition jeder Zeile einfach mehr 16-Bit Zahlen genommen. Es ist jetzt aber 18 Punkte breit:

```
0000011111111000 0000000000000000
0000011111111000 0000000000000000
1111111111111111 1100000000000000
1111111111111111 1100000000000000
0000011111111000 0000000000000000
0000011111111000 0000000000000000
   erste Zahl       zweite Zahl
```

Die zugehörigen Daten würden dann so aussehen:

```
USHORT Bild [] = {0x07f8,0x0000
                  0x07f8,0x0000
                  0xffff,0xc000
                  0xffff,0xc000
                  0x07f8,0x0000
                  0x07f8,0x0000};
```

Für Bilder, die aus mehreren Bitplanes bestehen, werden die einzelnen Bitplanes in der soeben beschriebenen Weise definiert und dann hintereinander in den Daten gespeichert. Ein zweifarbiger Streifen, der folgendermaßen aussieht:

```
  Bitebene 1
000000000000000
111111111111111

  Bitebene 2
111111111111111
000000000000000
```

Wird durch folgende Daten definiert:

```
USHORT Bild [] = {0x0000,   /* Bitplane 1  */
                  0xffff,
                  0xffff,   /* Bitplane 2  */
                  0x0000}
```

Es ist natürlich auch erlaubt, daß Teile des Bildes in beiden Bitebenen gleichzeitig liegen.

Der nächste Schritt nach dem Erstellen der Bildaten ist das Initialisieren der ***Image***-Struktur. Diese ist wie folgt definiert:

Abb 3.3 Die Image Struktur.

```
struct Image {

SHORT LeftEdge, TopEdge;
SHORT Width, Height, Depth;
SHORT *ImageData;
UBYTE PlanePick, PlaneOnOff;
struct Image *NextImage;

};
```

LeftEdge und ***TopEdge*** sind hier wie bei der ***Border***-Struktur die x- und y-Offsets. Die Breite und Höhe des Bildes in Pixels wird in ***Width*** und ***Height*** übergeben. ***Depth*** bestimmt die Tiefe, also die Anzahl der Bitebenen. In ***ImageData*** müssen Sie die Adresse der Bilddaten hineinschreiben. Diese müssen im CHIP-Ram liegen! Die nächsten beiden Felder, ***PlanePick*** und ***PlaneOnOff***, sind für die Farbwahl zuständig. Das erste besagt, welche Bitebenen des Bildschirms für die Darstellung des Images verwendet werden, das zweite, welche Farbe die Punkte bekommen, die in den Bildaten nicht gesetzt sind. Um das Kreuz von Abb.3.2 in Farbe 1, seinen Hintergrund in Farbe 3 zu zeichnen, müssen Sie also für ***PlanePick*** 1, für ***PlaneOnOff*** 3 angeben. Mit Hilfe von ***PlaneOnOff*** können Sie auch, ohne irgendwelche Bildaten ein gefülltes Rechteck einer beliebigen Farbe erzeugen. Dazu setzen Sie ***ImageData*** auf Null, ***Width*** und ***Height*** auf die gewünschte Größe des Rechtecks und tragen seine Farbe in ***PlaneOnOff*** ein. Da keine Bildaten existieren, besteht für Intuition das gesamte Bild aus Nullen, und wird daher in der gewünschten Farbe dargestellt.

Um das durch die *Image*-Struktur beschriebene Bild darzustellen, müssen Sie die *DrawImage*-Prozedur so aufrufen:

DrawImage(RPort,ImagePtr,x,y);

Wie bei der *DrawBorder*-Routine ist *RPort* hier die Adresse eines Rastports. *ImagePtr* ist der Zeiger auf die *Image*-Struktur, *x,y* die Koordinaten, an denen das Bild erscheinen soll.

In dem nun folgendem Beispielprogramm haben wir uns bemüht, Ihnen vor allem die Bedeutung von *PlanePick* und *PlaneOnOff* klarzumachen. Am Anfang wird ein Bild eines Computers, mit der Tiefe 1 definiert. *PlanePick* wird auf 1, *PlaneOnOff* auf 0 gesetzt und das Bild wird gezeichnet. Es erscheint in der Vordergrundfarbe. Hiernach wird *PlanePick* eine 2 zugewiesen und das Bild wird nochmal ausgegeben. Es erscheint diesmal in der Farbe Nummer 2. Als letztes wird dann noch *PlaneOnOff* auf 1 gesetzt. Die Farbe des Computers ändert sich hierbei zwar nicht, sein Hintergrund nimmt aber die Farbe 1 an. Jedesmal, wenn Sie die linke Maustaste drücken, gibt das Programm ein neues Bild aus. Um es abzubrechen, klicken Sie das Close-Gadget an.

Beachten Sie auch, daß die Bilddaten vor dem Zeichnen in einen allozierten CHIP-Ram Breich kopiert werden.

Programm 3.2 Image.

```
#include "Display.h"
#include "intuition/intuition.h"
#include "exec/types.h"
#include "exec/memory.h"

struct Window *Window;
struct Screen *Screen;

/*  Bilddaten   */
USHORT Data [] = { 0x3ffc, /*  001111111111111100  */
                   0x300c, /*  001100000000001100  */
                   0x300c, /*  001100000000001100  */
                   0x300c, /*  001100000000001100  */
                   0x300c, /*  001100000000001100  */
                   0x300c, /*  001100000000001100  */
                   0x3ffc, /*  001111111111111100  */
                   0x0ff0, /*  000011111111110000  */
                   0xffff, /*  111111111111111111  */
                   0xfff1, /*  111111111111110001  */
                   0xffff, /*  111111111111111111  */
                   0x4002};/*  010000000000000010  */


struct Image Bild  = {0, 0,
                      16,12,
                      1,
                      0x1,0x0,
                      NULL};

main ()
{
 APTR   IData;
 USHORT code;
 SHORT  x, y;
 ULONG  Class;
```

```
OpenIntui();

/*  Einen low-res Screen öffnen.  */
Screen = (struct Screen *)MakeScr(0,0,320,250,
                       "Low-res",2,NULL,NULL,NULL);
if (Screen == NULL)
 exit(FALSE);

/*  Ein GZZ-Fenster auf dem neuen Screen öffnen  */
Window = (struct Window *)MakeWindow(0,0,320,250,
                           300,250,"Image-Beispiel",
            WINDOWCLOSE | GIMMEZEROZERO | ACTIVATE,
            MOUSEBUTTONS | CLOSEWINDOW,Screen);

if(Window == NULL)    /*  Fehler beim Öffnen ?  */
 exit(FALSE);

/*  CHIP-Memory für Bildaten allozieren  */
IData = AllocMem(sizeof(Data),MEMF_CHIP | MEMF
                                          PUBLIC);
if (IData == NULL)   /*  Fehler beim Allozieren  */
 exit(FALSE);

/*  und Daten dorthin kopieren.  */
CopyMem(&Data[0],IData,sizeof(Data));

/*  Zeiger in der Image-Struktur auf Bildaten setzen
Bild.ImageData = IData;

/*  Image Zeichnen  */
DrawImage(Window->RPort,&Bild,120,120);
Bild.PlanePick = 0x02;

/*  und Image zeichnen.  */
DrawImage(Window->RPort,&Bild,140,120);

/*  Andere Bitplane für "Nullpunkte" wählen  */
Bild.PlaneOnOff = 0x01;
```

```
/* und Image zeichnen. */
DrawImage(Window->RPort,&Bild,160,120);

/* Alte Bitplane für "Nullpunkte" wählen. */
Bild.PlaneOnOff = 0x00;

while (WaitEvent(Window,&code) != CLOSEWINDOW)
{
 GetMouse(Window,&x,&y);
 DrawImage(Window->RPort,&Bild,x,y);
}
 /* Datenspeicher wieder freigeben und alles
                                      schließen. */
FreeMem(IData,sizeof(Data));
Class = WaitEvent(Window,code);
CloseWindow(Window);
CloseScreen(Screen);
}
```

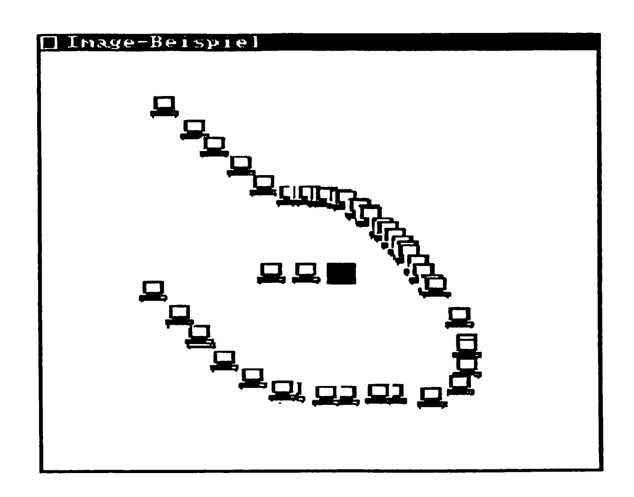


Bild 3.2 - Eine Beispielausgabe des Programms Image

Kapitel 4

==
Farbeinstellung und einfache Grafikprozeduren
==

Nachdem wir uns in den ersten drei Kapiteln mit ***Intuition*** befaßt haben, kommen wir nun endlich dazu, die eigentlichen Grafikroutinen zu betrachten. Wir wollen hier zuerst am Beispiel einiger einfacher Zeichenprozeduren zeigen, wie die Nummer der Zeichenfarbe, ihre RGB-Zusammensetzung, der Zeichenmodus und das für die Linie verwendete Muster eingestellt werden können. Dabei werden Sie die beiden grundlegenden Strukturen der Graphics-Library: ***RastPort*** und ***View Port*** sowie ihre Bedeutung kennenlernen. Vor allem der ***Rastport*** wird Ihnen bei der Grafikprogrammierung noch sehr oft begegnen. Beachten Sie, daß Sie die Files grahics/graphics.h und eventuell graphics/gfxmacros.h mittels ***include*** in Ihr Programm einbinden müssen, um auf die hier verwendeten Prozeduren und Strukturen zugreifen zu können.

1 Der Rastport

Die Screens und Fenster sind zwar vom Standpunkt des Benutzers aus die elementaren Anzeigenelemente, vom System aus gesehen stehen diese aber auf einer ziemlich hohen Stufe, werden also unter Verwendung anderer grafischer Routinen und "Bauteile" von Intuition aufgebaut.

Damit man die Zeichenroutinen sowohl in einem Fenster als auch in einem Screen oder einem selbsterzeugten Display verwenden kann, wird diesen nicht ein Zeiger auf ein Fenster, sondern die Adresse einer allgemeineren Struktur, die Bestandteil jeder Anzeige ist, übergeben.

Es handelt sich dabei um die **RastPort**-Struktur. Diese bestimmt sozusagen alle Eigenschaften des "Zeichenstiftes", also z.B. die Vorder- und Hintergrundfarbe, den Zeichenmodus, die Schriftart etc., die in einer Anzeige verwendet wird. Die Adresse des zu einem Fenster oder Screen gehörenden Rastports, können Sie wie in Kapitel 3 beschrieben, der ***Screen-*** bzw. ***Window***-Struktur entnehmen.

Da ***RastPort*** sehr viele Felder enthält, die nur für das System von Bedeutung sind, verzichten wir hier auf die vollständige Auflistung dieser Struktur und gehen nur auf die wichtigsten Felder ein.

***BitMap:** Zeiger auf die Bitmap, (also den "Bildspeicher") die Anzeige, zu der dieser Rastport gehört.

***AreaPtrn:** Das Muster, das zum Füllen von Flächen verwendet wird.

FgPen,BgPen: Die Vorder- und Hintergrundfarben, die zum Zeichnen benutzt werden.

AOlPen: Bestimmt die Umrandungsfarbe für gefüllte Flächen.

DrawMode: Der Zeichenmodus.

LinePtrn: Ein Muster für die Linien (z.B.gestrichelt).

CR cp_x,cp_y: Die momentanen Koordinaten des Grafikcursors.

***CR Font:** Der in diesem Rastport verwendete Schriftfont.

TxWidth,TxHeight,TxSpacing: Diese Felder geben die Breite, Höhe und den Abstand zwischen

den Zeichen des aktuellen Fonts an. Der Zeichenabstand kann durch verändern des *TxSpacing*-Wertes verändert werden, während die anderen Werte nur gelesen werden können.

Der obigen Auflistung können Sie entnehmen, wie die zum Zeichnen verwendete Farbe gespeichert wird. Um diese zu verändern, können Sie auf die ***SetAPen***-Prozedur zurückgreifen. Ist ***Rast*** ein Zeiger auf einen ***Rast Port***, so kann die dort gültige Vordergrundfarbe so auf 2 gesetzt werden:

SetAPen(Rast,2);

Analog kann die Hintergrundfarbe mit

SetBPen(Rast,Farbe);

modifiziert werden, wobei ***Farbe*** natürlich ein gültiger Farbwert sein muß. Beachten Sie, daß die Veränderung der Hintergrundfarbe nicht die Veränderung des Hintergrundes auf dem Bildschirm zur Folge hat. Es wird nur festgelegt, daß ab sofort, überall da, wo irgend etwas mit der Hintergrundfarbe gezeichnet oder gefüllt werden soll, der neue Farbwert verwendet wird. Wann dies der Fall ist, wird weitgehend durch den Zeichenmodus bestimmt.

Dabei gibt es vier Modi:

(1) JAM1: Dies ist der Standardmodus. Es wird zum Zeichnen nur die Vordergrundfarbe gebraucht.

(2) **JAM2:** Dieser Modus ist vor allem für die Linien- und Füllmuster wichtig, auf die wir noch zu sprechen kommen. Es werden die gesetzten Punkte des Musters mit der Vorder-, die nicht gesetzten mit der Hintergrundfarbe gezeichnet.

(3) **Complement:** In Diesem Modus wird jeder Punkt in der Farbe gezeichnet, die dem binären Komplement seiner bisherigen Farbnummer entspricht. Hatte ein Punkt also in einem Screen der Tiefe 2 bisher die Farbe Nummer 2 (binär 10), dann wird er nun mit der Farbe 1 (binär 01) gezeichnet. In einem Screen der Tiefe 5 würde er aber in der Farbe 13 (binär 1101 als das Komplement von 0010 = 2) dargestellt werden.

(4) **Inversevid:** Ist vorwiegend bei der Textdarstellung von Bedeutung. In Verbindung mit JAM1 bewirkt diser Modus, daß der Hintergrund des Zeichens in der Vordergrundfarbe dargestellt wird, während das Zeichen selbst durchsichtig ist. Der Unterschied bei Verwendung zusammen mit ***JAM2*** liegt darin, daß dann das Zeichen in der Hintergrundfarbe erscheint.

Der Zeichenmodus wird über Flags, die entsprechend ***JAM1***, ***JAM2***, ***COMPLEMENT***, und ***INVERSEVID*** heißen, mit Hilfe der **SetDrMd**-Prozedur z.B. wie folgt:

SetDrMd(Rast,JAM2);

eingestellt. Direkte Zugriffe auf die Datenstrukturen sollten so weit wie möglich vermieden werden, um das

Programm auch zu eventuellen späteren Systemversionen, bei denen diese vieleicht modifiziert werden, kompatibel zu halten.

2. Punkte und Linien

Punkte können an beliebiger Bildschirmposition mit der Prozedur ***WritePixel*** mit der aktuellen Farbe und den aktuellen Zeichenmodus gezeichnet werden.

Als Parameter müssen Sie nur den Zeiger auf den ***Rastport*** und die Koordinaten übergeben. Ein Aufruf kann also so aussehen:

WritePixel(Rast,10,10);

Oft ist es wichtig zu wissen, welche Farbe ein bestimmter Punkt hat. Für diese Situationen gibt es in der Graphics-Library die Routine ***ReadPixel***, die sozusagen die Umkehrung von ***WritePixel*** ist, also beim Aufruf die Farbe des Punktes an der angegebenen Position als Funktionsergebnis vom Typ int liefert.

Linien können mit Hilfe der Grafikprozeduren nur ab der aktuellen Position des Grafikcursors gezeichnet werden. Diese ist, wie dem vorangegangenen Abschnitt zu entnehmen ist, in der ***RastPort*** - Struktur gespeichert. Um den Grafikcursor neu zu positionieren müssen Sie die Prozedur ***Move*** wie folgt aufrufen:

Move(Rast,x,y);

Dabei ist ***Rast*** ein Zeiger auf einen Rasport und ***x*** und ***y*** die gewünschten Koordinaten. Eine Linie von der auf diese Weise bestimmten Position zu irgendeiner anderen Stelle auf dem Bilschirm kann dann mit der ***Draw***-Prozedur dargestellt werden. Diese braucht als Eingabe wieder die Rastportadresse und die Koordinaten des Zielpunktes. Um also eine Linie von (10,10) nach(100,100) zu zeichnen, müssen Sie so vorgehen:

Move(Rast,10,10);
Draw(Rast,100,100);

Für viele Anwendungen ist die Möglichkeit, Linien mit verschiedenen Mustern zu zeichnen, also z.B.gestrichelt oder punktiert, sehr interessant. Der Befehl ***SetDrPt*** erlaubt Ihnen, ein 16 Punkte langes Muster einem Rastport zuzuordnen. Dieses wird dann bis auf weiteres für alle in diesem Rastport gezeichneten Linien verwendet. Das Muster wird in einem SHORT-Wert gespeichert, dessen Bits den ein oder ausgeschalteten Punkten entsprechen. Die Linie wird dann als Aneinanderreihung solcher 16 Punkte langen Abschnitte gezeichnet. Einer gestrichelten Linie, die aus diesen 16-Punkte Abschnitten besteht:

1111111100000000

(Einsen stehen für gesetzte Punkte) entspricht dem hexadezimalen Wert ff00. Der dazugehörende Aufruf von SetDrPt sieht dann so aus:

SetDrPt (Rast, 0xff00);

Um wieder noramle Linien zeichnen zu können, rufen Sie ***SetDrPt*** mit -1 als Musterwert auf.

In dem nun folgenden Programm können Sie sehen, wie Linien mit verschiedenen Farben, Mustern und Zeichenmodi erzeugt werden können. In der ersten *for*-Schleife wird jedesmal, bevor eine Linie gezeichnet wird, eine neue Vordergrundfarbe gewählt. In der zweiten wird bei jedem Durchlauf ein neues Linienmuster gewählt, und dann eine Linie zuerst mit dem *JAM1*- und dann mit dem *JAM2*-Modus gezeichnet. Wie Sie sehen, werden beim *JAM2* Modus die in dem Muster auf 0 gesetzten Punkte nun mit der vorhin gewählten Hintergrundfarbe gezeichnet. Probieren Sie doch mal aus, was passiert, wenn sie die anderen Zeichenmodi, oder deren Mischung an dieser Stelle verwenden.

Programm 4.1 Draw

```
#include "Display.h"
#include "intuition/intuition.h"
#include "exec/types.h"
#include "graphics/gfx.h"
#include "graphics/gfxmacros.h"

struct Window *Window;
struct Screen *Screen;
struct RastPort *Rast;

/*  Einige Linienmuster definieren.  */
SHORT  LPattern [] =
                    { 0xff00,  /* 1111111100000000  */
                      0xf0f0,  /* 1111000011110000  */
                      0xcccc,  /* 1100110011001100  */
                      0xaaaa,  /* 1010101010101010  */
                    };

main()
{
 USHORT code;
```

```
SHORT  i;

OpenIntui();
OpenGfx ();

/*  Einen lowres Screen der Tiefe 4 (16 Farben)
                                          öffnen.  */
Screen = (struct Screen *)MakeScr(0,0,320,250,
                          "Linien",4,NULL,NULL,NULL);
if (Screen == NULL)  /*  Fehler beim Öffnen.  */
 exit(FALSE);

 /*  Ein GZZ-Fenster auf dem neuen Screen öffnen  */
Window = (struct Window *)MakeWindow(0,0,320,250,
                          320, 250,"Linien-Beispiel",
             WINDOWCLOSE | GIMMEZEROZERO | ACTIVATE,
                                CLOSEWINDOW,Screen);

if(Window == NULL)      /*  Fehler beim Öffnen ?  */
 exit(FALSE);

/* Die Adresse des Rastports des Fensters
                                       ermitteln. */
Rast = Window->RPort;

/*  15 Linien mit verschiedenen Farben zeichnen   */
for(i = 1; i < 16; i++)
{
SetAPen(Rast,i);     /* Neue Farbe auswählen.      */
Move(Rast,10,i*5);   /* Grafikcursor positionieren.*/
Draw(Rast,300,i*5);  /* Linie zeichnen.            */
}
/*  Farbe 1 als Vorder- und 2 als Hintergrund
                                        wählen.  */
SetAPen(Rast,1);

/*  Und 5 neue Linien mit verschiedenen Mustern und
                                    Zeichenmodi.  */
```

```
for(i = 1; i < 6; i++)
{
  SetDrMd(Rast,JAM1);
                /*  Normaler Zeichenmodus.          */
  SetDrPt(Rast,LPattern[i-1]);
              /*  Neues Muster auswählen.           */
  Move(Rast,10,i*10+100);    /*  Grafikcursor
                                    positionieren. */
  Draw(Rast,300,i*10+100);   /*  Linie zeichnen.    */
  SetDrMd(Rast,JAM2);        /*  Neuer Zeichenmodus*/
  Move(Rast,10,i*10+5+100); /*  Grafikcursor
                                    positionieren. */
  Draw(Rast,300,i*10+5+100);/*  Linie zeichnen.    */
}
/*  Auf Close Gadget warten  */
i = WaitEvent(Window,&code);

/*  und alles schließen  */
CloseWindow(Window);
CloseScreen(Screen);
}
```

Bild 4.1 - Die Ausgabe des Programms Draw

3 Rechtecke, Kreise und Ellipsen

Für komplexere Grafikobjekte genügt es allerdings nicht mehr, Punkte und Linien zu zeichnen. Da alle aus Linien bestehenden Figuren wie Rechtecke und Dreiecke mit Hilfe mehrere Aufrufe der ***Draw***, oder wie wir später sehen werden, eines Aufrufs der ***PolyDraw***-Prozedur, mühelos erzeugt werden können, bietet der Amiga keine eigenständige Prozedur zum Zeichnen dieser Objekte. Zum Zeichnen von Kreisen und Ellipsen gibt es dagegen die ***DrawCircle***- und ***DrawEllipse***-Routinen. ***DrawCircle***, das in "gfxmacros. h" definiert wird, benötigt als Eingabe die Adresse des Rastports, die Koordinaten des Kreismittelpunktes und den Radius. Ein Kreis mit dem Radius 50 wird also auf diese Weise an der Position (150,100) gezeichnet:

DrawCircle(Rast,150,100,50);

Leider beachtet diese Prozedur nicht die Auflösung des verwendeten Displays, so daß der Kreis nur auf einem lowres oder einem hires-interlace Screen wirklich rund ist.

Um eine beliebige Ellipse zu zeichnen, genügt ein folgender Aufruf von DrawEllipse:

DrawEllipse(Rast,x,y,rx,ry);

Dabei ist ***Rast*** ein Zeiger auf den Rastport, ***x,y*** die Koordinaten des Mittelpunktes und ***rx,ry*** der ***x***- bzw.y-Radius der Ellipse.

Das Zeichnen verschiedener Figuren wird in dem folgenden kurzen Programm vorgeführt. Es zeichnet zuerst konzentrische Kreise und Ellipsen mit verschiedenen Radien. Dann werden mittels einer dafür neu definierten Prozedur mehrere gegeneinander verschobene Rechtecke erzeugt. Beachten Sie, daß das Programm in einem Screen und nicht in einem Fenster arbeitet.

Programm 4.2 Figuren

```
#include "Display.h"
#include "intuition/intuition.h"
#include "exec/types.h"
#include "graphics/gfx.h"
#include "graphics/gfxmacros.h"

struct Screen *Screen;
struct RastPort *Rast;

main ()
{
 long j;
 SHORT i;

 OpenIntui();
 OpenGfx ();
 /*  Einen hi-res Screen der Tiefe 2 (4 Farben)
                                       öffnen.  */
 Screen = (struct Screen *)MakeScr(0,0,320,250,
                              "Figuren",2,NULL,NULL);
 if (Screen == NULL)   /*  Fehler beim Öffnen.   */
  exit(FALSE);
 /*  Die Adresse des Rastports des Screens
                                    ermitteln.  */
 Rast = &((*Screen).RastPort);

 /*  Die erste Figurenreihe zeichnen.   */
 for(i = 1; i < 5; i++)
```

```
 {
  DrawCircle(Rast,70,70,i*10);  /* Kreis zeichnen. */
  DrawEllipse(Rast,250,70,i*10,50-i*10);
                                /* Ellipse Zeichnen. */
 }
  /*  Und einige Vierecke zeichnen.  */
 for(i = 1; i < 10; i++)
   Rect(Rast,50+i*5,150+i*2,220+i*5,200+i*2);
 }
 /*  Abwarten und dann Screen schließen  */
 for(j = 1; j < 400000; j++);
 CloseScreen(Screen);
}
/*  Diese Prozedur zeichnet ein Rechteck  */
Rect(R,x1,y1,x2,y2)
APTR  *R;
SHORT x1, y1, x2, y2;
{
 Move(R,x1,y1);
 Draw(R,x2,y1);
 Draw(R,x2,y2);
 Draw(R,x1,y2);
 Draw(R,x1,y1);
}
```

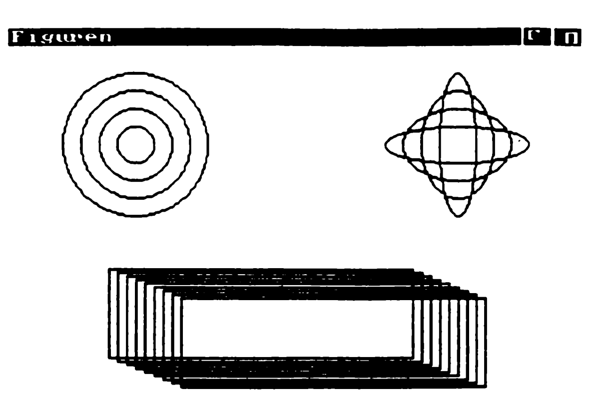


Bild 4.2 - Die Ausgabe Programms Figuren

4 Die RGB-Werte der Farben

Bisher sind wir nur in der Lage, durch Angabe einer Nummer, eine der maximal 32 Farben, die für den Screen voreingestellt sind, zu wählen. Der Amiga gibt Ihnen aber die Möglichkeit, eine aus einer Palette von 4096 verschiedenen Farben auszusuchen. Sie können diese normalerweise zwar nicht alle gleichzeitig benutzen, dafür aber mit der ***SetRGB4***-Prozedur die RGB-Anteile, der zur jeder Nummer gehörenden Farbe wählen (jeweils zwischen 0 und 15 liegende Rot-, Grün- und Blau-Anteile). Die Prozedur ***SetRGB4*** braucht neben der Nummer der Farbe und der neuen RGB-Werte einen Zeiger auf die ***ViewPort***-Struktur. Dies ist die grundlegende Struktur jeder eigenständigen Anzeige, also eines Screens oder eines selbsterzeugten Displays. So wie die Eigenschaften eines "Zeichenstiftes" in der ***RastPort***-Struktur gespeichert sind, können Sie die wichtigsten Eigenschaften des Displays dem ***Viewport*** entnehmen. Sie werden später noch sehen, wie man mit Hilfe eines neuen Viewports diverse unter Intuition nicht zulässige Anzeigen realisieren kann, und dabei auch den genauen Aufbau der ***ViewPort***-Struktur und die Bedeutung ihrer Felder kennenlernen.

Um nun beispielsweise der Farbe Nummer 1 eine Mischung aus 7 Anteilen Grün und 7 Anteilen Blau zuzuordnen rufen Sie ***SetRGB4*** folgendermaßen auf:

SetRGB4(View,1,0,7,7);

View ist selbstverständlich ein Zeiger auf die ***ViewPort***-Struktur. Je nachdem, ob Sie direkt in einem Screen, oder in einem Fenster arbeiten, müssen Sie

unterschiedlich vorgehen, um diesen zu ermitteln. Um den Viewport eines Fensters zu finden, können Sie sich der Intuition-Prozedur ***ViewPortAdress*** bedienen, die als Eingabe einen Zeiger auf ein Fenster braucht, und als Ergebnis die gesuchte Adresse des Viewports liefert. Der ***Viewport*** eines Screens ist direkt in der ***Screen***-Struktur zu finden. Seine Adresse kann also wie folgt z.B. in die Variable View eingelesen werden:

View = &((*Screen).Viewport);

Dabei wird natürlich vorausgesetzt, daß ***Screen*** ein Zeiger auf den Screen ist. Beachten Sie auch, daß sich die gesamte ***ViewPort***-Struktur, nicht nur Ihre Adresse in der ***Screen***-Struktur befindet.

Die RGB-Werte für die Farben können selbstverständlich nicht nur verändert, sondern auch jederzeit gelesen werden. Notwendig ist das beispielsweise, um die von vielen Zeichenprogrammen bekannte Color-Cycle-Funktion, die die Werte der Farbregister zyklisch verändert, zu verwirklichen.

Die gewünschten Werte werden Ihnen von der Prozedur ***GetRGB4*** geliefert. Sie gibt Ihnen eine Zahl, in der die drei gesuchten Werte enthalten sind, zurück.

Falls Sie sich wundern sollten, wie das geht, dann denken Sie bitte daran, daß jeder Wert zwischen 0 und 15 liegt, also nur 4 Bit braucht, so daß eine 16 Bit-Zahl völlig ausreicht, um alle drei Komponenten zu speichern. Den Anteil einer bestimmten Farbe kann man mit Hilfe einer AND-Verknüpfung der erhaltenen Zahl, mit einem Wert, der den zu dieser Farbe gehörenden Bits entspricht, erhalten. Die untersten 4 Bit sind für die Farbe Blau reserviert, die nächsten 4 für

Grün und die darüberliegenden für Rot. Die obersten 4 Bits sind bedeutungslos. Daraus folgt, daß man den Roten Anteil durch eine AND-Verknüpfung mit hexadezimal 0f00 (dezimal 3840), den Grünen mit 00f0 (240) und den Blauen mit 000f (15) bekommen kann. Hier ein Beispiel:

```
View   = ViewPortAddress(Window);
Colors = GetRGB4(CMap,1);
Rot    = Colors & 3840;
Gruen  = Colors & 240;
Blau   = Colors & 15;J
```

In diesem Beispiel müssen *Window* ein Zeiger auf ein Fenster und *CMap* die Adresse einer *ColorMap*-Struktur sein. Die letztere können Sie aus der *ViewPort*-Struktur folgendermaßen ermitteln:

```
CMap = View->ColorMap;
```

Wie das in der Praxis funktioniert, können Sie sich in dem folgenden Programm ansehen. In einer *for*-Schleife wird dort der Blauanteil aller Farben des Screens auf Werte von 0 bis 15 gesetzt. Dann werden mit jeder Farbe fünf konzentrische Kreise mit immer größer werdenden Radien gezeichnet, so daß durch die zunehmende Helligkeit der Eindruck eines dreidimensionalen Tunnels entsteht. Die Adresse des Viewports wird hier nicht mit der *ViewPortAdress*-Prozedur, sondern über den zu dem Fester gehörenden Screen ermittelt. Zum Schluß wird wieder in einer *for*-Schleife immer wieder der Blauanteil von zwei benachbarten Farbregistern vertauscht, so daß der schon erwähnte Color-Cycle Effekt entsteht. Da die Rot und Grün Anteile aller Farben null sind, wäre hier die AND Ver-

knüpfung nicht notwendig, um den Blauanteil zu erhalten. Wir haben sie aber trotzdem zur Demonstration verwendet.

Programm 4.3 ColorCycle

```
#include "Display.h"
#include "intuition/intuition.h"
#include "exec/types.h"
#include "graphics/gfx.h"J
#include "stdio.h"

struct Screen   *Screen;
struct Window   *Window;
struct RastPort *Rast;
struct ViewPort *View;
struct ColorMap *CMap;

main ()
{

 long j, Color;
 SHORT i, k, *CAdr;

 OpenIntui();
 OpenGfx ();

 /*  Einen low-res Screen der Tiefe 4 (16 Farben)
                                          öffnen.  */
 Screen = (struct Screen *)MakeScr(0,0,320,200,
                               "Figuren",5,NULL,NULL);
 if (Screen == NULL)       /*  Fehler beim Öffnen.  */
  exit(FALSE);
```

```
/*  Fenster auf dem neuen Screen Öffnen.  */
Window = (struct Window*)MakeWindow(0,0,320,200,
                        0,0,"Hallo",SMART_REFRESH|
                           BORDERLESS,NULL,Screen);
if(Window == NULL)         /*  Fehler beim Öffnen.  */
  exit(FALSE);

/*  Adresse des Rastports, des Viewports und der
                                      ColorMap.  */
Rast = Window->RPort;
View = &((*Screen).ViewPort);
CMap = View ->ColorMap;

/*  Farbregister 2 bis 15 initialisieren.  */
SetRGB4(View,0,0,0,0);      /*  Hintergrundfarbe
                                        schwarz  */
for(i = 1; i < 16; i++)
  SetRGB4(View,i,0,0,i);
/*  80 Konzentrische Kreise zeichnen.  */
for(i = 2; i < 16; i++)
{
  SetAPen(Rast,i);
                   /*  Neue Farbe wählen und        */
  for(k = 1; k < 6; k++)
                   /*  5 Kreise in dieser  Farbe.  */
  {
    DrawCircle(Rast,160,100,i*5+k);
  }
}

/*  Farben 50 mal vertauschen.  */
for(i= 2; i < 50; i++)
{
  for( k = 1; k < 16; k++)
  {
    Color = GetRGB4(CMap,k)&15;
    SetRGB4(View,k,0,0,GetRGB4(CMap,(k+1)%14+2)&15);
    SetRGB4(View,(k+1)%14+2,0,0,Color);
```

```
        }
        for(j = 1; j < 30000; j++);
    }
    CloseWindow(Window);
    CloseScreen(Screen);

}
```

Kapitel 5

==
Polygone, gefüllte Flächen und Füllmuster
==

Nach den doch noch ein bißchen primitiven Zeichenroutinen, die Sie im vierten Kapitel kennengelernt haben, wollen wir uns hier einem komplizierteren, aber dafür auch interessanteren Thema zuwenden. Dabei handelt es sich um das Zeichnen von Vielecken und gefüllten Flächen. Dabei gibt es noch eine Besonderheit: die Möglichkeit, für diese Flächen verschiedene Muster bzw. Farbmuster zu verwenden. In diesem Zusammenhang wird ein include-File vorgestellt, das Ihnen bei der Initialisierung der von den Füllroutinen benötigten Strukturen (***TempRas*** und ***AreaInfo***) behilflich sein soll. Er erhält auch eine Prozedur, die Ihnen beim Zeichnen von gefüllten Polygonen einige Arbeit abnehmen wird.

1 PolyDraw: Zeichnen von Vielecken

Gute Grafik läßt sich nur schwer aus Rechtecken und Ellipsen allein aufbauen. Mit der ***DrawBorder***-Prozedur haben Sie bereits eine Möglichkeit kennengelernt, beliebige Vielecke bzw. Linienzüge zu zeichnen. Eine einfacherere Möglichkeit stellt die ***PolyDraw***-Prozedur der Graphics-Library dar. Diese braucht als Eingabe die Anzahl der Ecken, sowie einen Zeiger auf den Rastport und auf einen Speicherbereich, in dem die Koordinaten der Eckpunkte des zu zeichnenden Vielecks als Zahlenpaare gespeichert sind. Ein Aufruf muß also wie folgt aussehen, wenn über den ***n*** Ecken der in ***Koordinaten*** gespeicherten Werten ein Polygon gezeichnet werden soll:

PolyDraw(Rast,n,&Koordinaten[0]);

Am besten ist es, wenn Sie ihre Koordinaten in einem Array von *SHORT*-Zahlen speichern und deren Adresse dann an diese Prozedur weitergeben. Da Sie die Anzahl der Ecken extra angeben, können Sie wahlweise auch nur einen Teil des Vielecks zeichnen lassen. Zu beachten ist lediglich, daß die Prozedur das Polygon nicht von selbst abschließt, d.h. daß Sie, um eine geschlossene Figur zu erhalten, die Koordinaten des ersten Punktes auch am Ende als Koordinaten des letzten Punktes angeben müssen. Wichtig ist auch, daß vor dem Zeichnen des Polygons der Grafikcursor an die Position der ersten Ecke gebracht wird, da ansonsten diese mit seiner aktuellen Position verbunden wird.

Als Beispiel für die Anwendung der *PolyDraw*-Prozedur folgt ein Programm, das dem Benutzer die Eingabe der Eckkoordinaten durch Klicken der Maus erlaubt. Die Position der Maus beim Drücken der linken Taste wird in dem Array *Polygon*, das später an *PolyDraw* übergeben wird, gespeichert, und legt somit die Position einer Ecke fest. Sobald die rechte Taste gedrückt wird, wird die Eingabe beendet, der Grafikcursor positioniert und das Polygon gezeichnet. Durch Anfügen des ersten Koordinatenpaares am Ende des Arrays wird das Polygon abgeschlossen.

Programm 5.1 Polygon

```
#include "Display.h"
#include "intuition/intuition.h"
#include "exec/types.h"
#include "graphics/gfx.h"

struct Screen   *Screen;
struct Window   *Window;
struct RastPort *Rast;
```

```
main ()
{
 SHORT  x, y, Corners, Polygon[100];
 USHORT code;
 ULONG  Class;

 OpenIntui();
 OpenGfx ();

 /*  Einen hi-res Screen der Tiefe 2 (4 Farben)
                                           öffnen.  */
 Screen = (struct Screen *)MakeScr(0,0,640,250,
                             "Polygon",2,HIRES,NULL);
 if (Screen == NULL)       /*  Fehler beim Öffnen.  */
  exit(FALSE);

 /*  Rahmenloses Fenster mit Close Gadget und  IDCMP-
                                      Port öffnen.  */
 Window = (struct Window *)MakeWindow(0,0,640,250,0,
                                       0,"Polygon",
   SMART_REFRESH| BORDERLESS | WINDOWCLOSE | RMBTRAP,
                   CLOSEWINDOW|MOUSEBUTTONS,Screen);
 if(Window == NULL)
  exit(FALSE);             /*  Fehler beim Öffnen.  */

 /*  Adresse des Rastports.  */
 Rast = Window->RPort;
 /*  Ecken einlesen und Polygon zeichnen.  */
 Corners = 0;          /*  Eckenzahl initialisieren.  */
 do {
  Class = WaitEvent(Window,&code);
            if((code&MENUUP) == MENUUP)
                                 /*  Ende der Eingabe ? */
   break;              /*  Ja -> Schleifenabbruch.  */
  if((code&SELECTUP) == SELECTUP)
                                 /*  Ecke eingegeben ?  */
  {
   GetMouse(Window,&x,&y);
```

```
            /*  Ja -> Mauskoordinaten lesen,           */
    WritePixel(Rast,x,y);
            /*  Punkt makieren,                         */
    Polygon[2*Corners] = x;
            /*  Koordinaten merken                      */
    Polygon[2*Corners+1] = y;
    Corners++; /*  und Eckenanzahl erhöhen,             */
   }
  } while (Corners < 50);

  /*  Das Polygon zeichnen.  */
  Polygon[2*Corners] = Polygon[0];
                          /*  Polygon abschließen      */
  Polygon[2*Corners+1] = Polygon[1];
  Move(Rast,Polygon[0],Polygon[1]);
  PolyDraw(Rast,Corners+1,&(Polygon[0]));

  /*  Auf Close Gadget warten und alles schließen.     */
  while((WaitEvent(Window,&code) != CLOSEWINDOW));
  CloseWindow(Window);
  CloseScreen(Screen);
}
```

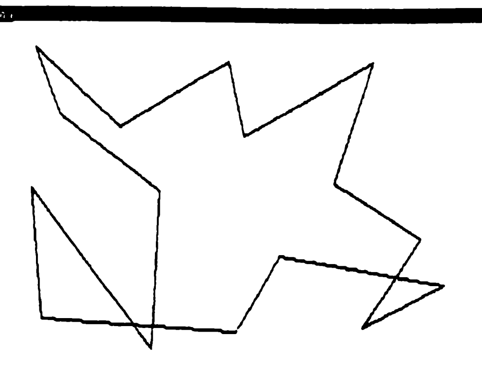


Bild 5.1 -Eine Beispielausgabe des Programms Polygone

2 Flächen füllen: TempRas und AreaInfo

Die Befehle zur Erzeugung gefüllter Flächen sind in sich relativ simpel, vor ihrer Anwendung müssen aber in der Regel einige Vorbereitungen getroffen werden. Der Grund dafür ist darin zu suchen, daß das System um komplizierter geformte Flächen korrekt und schnell zu füllen, zusätzlichen freien Speicher als Zwischenablage für Bilddaten und Eckkoordinaten benötigt. Die Verfügbarkeit, Größe und Adresse dieses Speichers entnehmen die Grafikprozeduren den Datenstrukturen ***TempRas*** und ***AreaInfo***, deren Adressen in der ***RastPort***-Struktur eingetragen werden müssen.

Um diese Strukturen initialisieren zu könnnen, müssen Sie zuerst den benötigten Speicher reservieren. Der Bildatenzwischenspeicher, der durch ***TempRas*** beschrieben wird, muß im CHIP-Ram liegen, und groß genug sein, um den gesamten in jeweils einer Bitplane liegenden Teil der zu füllenden Fläche komplett aufzunehmen. Beachten Sie dabei, daß sich die Größe des für eine Figur benötigten Speichers aus dem Produkt ihrer maximalen Höhe und Breite ergibt. Für ein Dreieck der Höhe 90 Pixel un Breite 70 Pixel brauchen Sie also einen Speicher von 90x70 Bit, was aufgerundet (Speicherplatz wird ja nur in ganzen Bytes verwaltet!) 12x10 = 120 Bytes = 6x5 Prozessorwörtern, entspricht. Um eine ***TempRas***-Struktur zu initialisieren, muß, nach dem der Speicher alloziert wurde, die ***InitTempRas***-Routine der Graphics-Library folgendermaßen aufgerufen werden:

InitTmpRas(&TRas,Memory,Size)

Memory ist dabei der Zeiger auf den allozierten Speicherbereich und *Size* seine Größe. In *TRas* wird ein Zeiger auf die initialisierte *TempRas*-Struktur zurückgegeben (*TRas* muß also vom Typ *TempRas* sein). Diesen müssen Sie, um den so erzeugten *TempRas* einem Rastport zur Verfügung zu stellen, nur noch wie folgt in die entsprechende *RastPort*-Struktur eintragen:

```
Rast->TempRas = TRas;
```

Denken Sie daran, daß Sie den Speicher später durch Deallozieren wieder freigeben und das *TempRas*-Feld des Rastports wieder auf NULL setzen.

Als nächstes muß die *AreaInfo*-Struktur erzeugt werden. Der dafür benötigte Speicher hängt davon ab, welche Füllroutine Sie gebrauchen wollen. Grundsätzlich gilt, daß pro Ecke der zu zeichnenden Figur 10 Bytes benötigt werden. Für ein Polygon mit 10 Ecken bräuchten Sie also einen 100 Byte großen Buffer. Die *AreaInfo*-Struktur wird nach der Allozierung des Speichers durch den folgenden Aufruf der *InitArea*-Prozedur initialisiert:

```
InitArea(&AInfo,Buffer,Size);
```

In den beiden Parametern *Buffer* und *Size* müssen Sie die Adresse des Buffers und die Anzahl der Ecken mal 2 angeben. In *AInfo* wird dann ein Zeiger auf die initialisierte *AreaInfo*-Struktur zurückgegeben, den Sie wieder in die *RastPort*-Struktur wie folgt eintragen müssen.

```
Rast->AreaInfo = &AreaI;
```

Wie Sie sehen, ist die Vorbereitung zum Füllen von Flächen doch relativ aufwendig. Um Ihnen zukünftig die damit verbundene Mühe zu ersparen, wird in

dem hier abgedruckten "include"-File die Prozedur ***NewArea*** definiert, die alle oben beschriebenen Maßnahmen für Sie erledigt. Zusätzlich beinhaltet dieser File auch noch eine Prozedur, die den von ***Tempras*** und ***AreaInfo*** belegten Speicher wieder freigibt (***CloseArea***) und eine, die ein gefülltes Polygon zeichnet (***PolyFill***). Auf die Funktionsweise und das Aufrufformat der letzten Prozedur werden wir in einem späteren Abschnitt zurückkommen. In ***NewArea*** wird eine neue Art Speicherplatz zu allozieren verwendet. Bei dieser Methode wird die Intuition-Routine ***AllocRemember*** aufgerufen, die den gewünschten Speicher reserviert und seine Größe in einer ***Remember-*** Struktur speichert. Neben der Größe und Art des benötigten Speichers müssen Sie der Routine auch noch die Adresse eines Zeigers auf eine solche Struktur übergeben. Falls dieser Zeiger NULL ist, wird dort die Adresse einer neuen ***Remember***-Struktur hinein geschrieben. Sie können später wenn Sie weitere Speicherbereiche allozieren, diesen Zeiger wieder verwenden, wodurch die neue ***Remember***-Struktur einfach an die alte "angehängt" wird. Der Vorteil dieses Verfahrens liegt darin, daß es möglich ist, mehrere Bereiche, die so alloziert wurden, zu einem späteren Zeitpunkt durch einen einzigen Aufruf von ***FreeRemember*** wieder freizugeben. Man braucht sich dabei nicht mal die Größe dieser Bereiche zu merken.

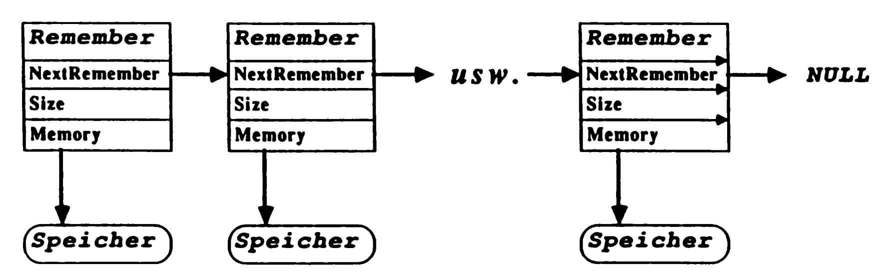


Bild 5.2 - Die interne Verwaltung der Rememberstrukturen

Programm 5.2 AreaExtras

```
#include "intuition/intuition.h"
#include "exec/types.h"
#include "exec/memory.h"
#include "graphics/gfx.h"

/*  Diese Prozedur initialisirt ein TempRas und
                                       ein AreaInfo   */
NewArea (Rast,Corners,RPointer)

SHORT Corners;APTR  *RPointer;
struct RastPort *Rast;

{
 static  struct TmpRas   TRast;
 static  struct AreaInfo AreaI;
 struct  Remember *Rm;
 APTR    TBuffer, ABuffer;
 long    PlaneSize;

 Rm = NULL;
 /*  CHIP-Memory für Temprast und AreaBuffer
                                        allozieren.  */
 PlaneSize = (Rast->BitMap->BytesPerRow)*(Rast->
                                         BitMap->Rows);
 TBuffer = AllocRemember(&Rm,PlaneSize,MEMF_CHIP |
                                          MEMF_PUBLIC);
 if (TBuffer == NULL) /*  Fehler beim Allozieren ? */
  exit(FALSE);
 ABuffer = AllocRemember(&Rm,Corners*10,MEMF_CHIP |
                                          MEMF_PUBLIC);
 if (TBuffer == NULL)  /* Fehler beim Allozieren ? */
  exit(FALSE);
 /*  TempRas und AreaInfo initialisieren,   */
 InitArea(&AreaI,ABuffer,2*Corners);
 InitTmpRas(&TRast,TBuffer,PlaneSize);
```

```
 /*  und seine Adresse im Rastport eintragen.  */
 Rast->TmpRas = &TRast;
 Rast->AreaInfo = &AreaI;
 *RPointer = Rm;
}

/* Diese Prozedur gibt den für TempRas und  AreaInfo
                                  nötigen Speicher frei  */
void CloseArea(Rast,RPointer)

APTR RPointer;
struct RastPort *Rast;

{
 FreeRemember(&RPointer,TRUE);  /*  Speicher
                                           freigeben.  */
 Rast->AreaInfo = NULL;
 /*  Eintrag im RastPort löschen.  */
 Rast->TmpRas   = NULL;

}
/*  Diese Prozedur erzeugt ein gefülltes Polygon.  */
void PolyFill(Rast,Polygon,Corners)

struct RastPort *Rast;
SHORT  *Polygon;
SHORT  Corners;
{
 SHORT i;
 APTR  Remember;

 /*  TempRas und AreaInfo initialisieren.  */
 NewArea(Rast,Corners,&Remember);

 /*  Grafikcursor zum Startpunkt, Ecken erzeugen und
                                      Polygon zeichnen.  */
 AreaMove(Rast,*Polygon,*(Polygon+1));
 for(i = 1; i < Corners; i++)
```

```
    AreaDraw(Rast,*(Polygon+2*i),*(Polygon+2*i+1));
  AreaEnd(Rast);

  /*  TempRas und AreaInfo löschen.  */
  CloseArea(Rast,Remember);
}
```

Wie dem Listing zu entnehmen ist, können Sie die *NewArea* und *CloseArea* Prozeduren folgendermaßen gebrauchen:

```
NewArea(Rast,Corners,RPointer);
CloseArea(Rast,RPointer);
```

Die Parameter haben dabei folgende Bedeutung:

Rast: Zeiger auf den Rastport.
Corners: Die Anzahl der Ecken.
RPointer: Die Adresse eines Zeigers auf eine *Remember*-Struktur.

Da man im voraus oft nicht genau sagen kann, wie groß die zu füllende Fläche ist, ist es am sinnvollsten, für den *TempRas* eine ganze Bitplane zu reservieren. Genau das tut auch das obige *AreaExtras* Programm, das die Größe der Bitplane wie folgt ermittelt:

```
b = Rast->BitMap->BytesPerRow;
h = Rast->BitMap->Rows;
PlaneSize = b*h;
```

Dabei wird über den in *Rast* enthaltenen Zeiger auf die *RastPort*-Struktur auf die *BitMap*-Struktur zugegriffen, wo die Anzahl der Bytes pro Zeile (*BytesPerRow*) und die Anzahl solcher Zeilen(*Rows*) gespeichert sind. Auf der Programmdiskette befindet sich des oben abgedruckte File in der Directory include unter dem Namen AreaExtras.h.

3 Die Füllprozeduren

Als erstes wollen wir in diesem Abschnitt die ***Flood***-Routine, die eine beliebige geschlossene Fläche um die angegebenen Koordinaten füllt, betrachten. Diese wird wie folgt

Flood(Rast, Mode, x, y);

aufgerufen und kann dazu verwendet werden, eine mit den schon beschriebenen Zeichenprozeduren erzeugte Figur nachträglich zu füllen. Dafür stehen Ihnen zwei Modi, die durch die ***Mode***-Variable ausgewählt werden können, zur Verfügung:

(1) outline-mode

Falls Sie diesen Modus wählen, werden alle Punkte um ***x,y,*** die nicht die Area-Outline Farbe haben, in der aktuellen Vordergrundfarbe gezeichnet.
Der Füllvorgang wird also dann gestoppt, wenn eine mit der Area-Outline Farbe gezeichnete Umrandung erreicht wird. Die Area-Outline Farbe in dem Rastport ***Rast*** kann mit Hilfe der ***SetOPen*** Routine so auf den in ***Color*** enthaltenen Wert gesetzt werden:

SetOPen(Rast,Color);

Dieser Modus wird eingeschaltet, wenn Mode den Wert 0 hat.

(2) color mode
In diesem Modus werden alle Punkte mit der aktuellen Vordergrundfarbe gefüllt, die um *x,y* liegen und die die gleiche Farbe besitzen, die der Punkt an der Position x,y hat. Der Füllvorgang wird hier beendet, wenn irgendeine geschlossenen Umrandung erreicht wird. Dies ist der am häufigsten verwendete Modus. Um ihn auszuwählen, setzen Sie Mode auf 1.

Im Outline-Modus können Sie die *Flood*-Routine verwenden, ohne vorher einen ***TempRas*** eingerichtet zu haben. Da das Füllen dabei extrem langsam vor sich geht, sollte man dies jedoch nach Möglichkeit vermeiden. Ein Versuch, ohne ***TempRas*** im color-Modus zu Füllen, bleibt entweder ohne Wirkung, oder endet mit einem Besuch beim Guru!

Wenn Sie ein Rechteck, eine Ellipse oder einen Kreis direkt ausgefüllt zeichnen wollen, dann können Sie sich der ***RectFill***, ***AreaEllipse***, oder ***AreaCircle***-Prozeduren bedienen. ***RectFill*** kann auch ohne einen ***TempRas*** gebraucht werden und erzeugt, wie der Name schon sagt, ein gefülltes Rechteck. Ein Rechteck, dessen linke obere Ecke bei (10,10) und rechte untere bei (110,110) liegen (Daraus ergibt sich Höhe =Breite = 100 Pixel), kann in dem durch ***Rast*** bestimmten Rastport so gezeichnet werden:

RectFill(Rast,10,10,110,110);

Für ***AreaEllipse*** und ***AreaCircle*** müssen wie im vorigen Abschnitt beschrieben, ein ***TempRas*** und ***AreaInfo*** initialisiert werden. Die Anzahl der Ecken ist dabei auf 4 zu setzen. Um eine Ellipse mit den Radien 90 (x-Radius) und 50 (y-Radius) und einen Kreis mit dem Radius 110 um den Punkt (320, 100) zu erzeugen muß man, nachdem die ensprechenden Vorbereitungen

(***TempRas***, ***AreaInfo***) getroffen wurden, die beiden Prozeduren so aufrufen:

AreaEllipse(Rast,320,100,90,50);<<Return>>AreaCircle (Rast,320,100,110);

AreaCircle ist in graphics/gfxmacros definiert.

Die schon im Zusammenhang mit ***Flood*** erwähnte ***Area-***Outline Farbe bietet zusätzlich noch die Möglichkeit, gefüllte Figuren mit einer Umrandung zu zeichnen. Dazu müssen Sie vor dem Aufruf der gewünschten Zeichenroutine mit Hilfe von ***SetOPen*** eine Farbe auswählen. Gleichzeitig wird dabei ein Flag im ***RastPort*** gesetzt, der dem System mitteilt, daß alle Flächen umrandet werden sollen. Um diesen Flag wieder zu löschen, können Sie das ***BNDRYOFF***-Macro benutzen, das in graphics/gfxmacros definiert ist. Der Aufruf sieht so aus:

BNDRYOFF(RAST);

Die Anwendung der soeben besprochenen Prozeduren können Sie sich an Hand des folgenden Programms klarmachen. Es eröffnet ein Fenster und zeichnet dort zwei sich überlappende Ellipsen. Danach wird die Area-Outline Farbe auf die Farbe der zweiten Ellipse gesetzt, und diese im Outline-Modus gefüllt. Obwohl sich die Ellipsen überlappen, wird die gesamte rechte Ellipse gefüllt, da das Füllen erst dort aufhört, wo sich die Area-Outline Farbe befindet. Würden Sie hier den color-Modus verwenden, dann würde die Ellipse nur bis zur Überlappung gefüllt werden. Als nächstes wird mit ***RectFill*** ein Viereck gezeichnet und dann mit Hilfe der in AreaExtras.h definierten ***NewArea***-Proze-

dur ein *TempRas* und ein *AreaInfo* für die nachfolgenden Flood und *AreaEllipse* Aufruf bereitgestellt. Beachten Sie um wieviel schneller das Füllen mit Flood nun geht. Zum Abschluß werden mit *CloseArea* der *TempRas* und *AreaInfo* wieder geschlossen.

Programm 5.3 Fill

```
#include "Display.h"
#include "AreaExtras.h"
#include "exec/types.h"
#include "intuition/intuition.h"
#include "graphics/gfx.h"
#include "graphics/gfxmacros.h"

struct Window   *Window;
struct RastPort *Rast;
struct Remember *Remember;

main ()
{

 USHORT code;
 ULONG  Class;

 OpenIntui();
 OpenGfx ();
 /*  Fenster mit Close- und Depth Gadget und  IDCMP-
                                   Port öffnen.  */
 Window = (struct Window *)MakeWindow(0,0,640,250,0,
                            0,"Fill",SMART_REFRESH|
          WINDOWCLOSE|WINDOWDEPTH,CLOSEWINDOW,NULL);
 if(Window == NULL)        /*  Fehler beim Öffnen.  */
   exit(FALSE);
 Rast = Window->RPort; /*  Adresse des Rastports.  */
```

```
/*  Zwei überschneidende Ellipsen  mit verschiedenen
                                   Farben zeichnen.   */
SetAPen(Rast,1);
DrawEllipse(Rast,220,50,150,40);
SetAPen(Rast,2);
                     /*  Neue Vordergrundfarbe !   */
DrawEllipse(Rast,440,50,150,40);
/*  Zweite Ellipse im Füllmodus 0 Füllen.   */
SetOPen(Rast,2);
                 /*  Neues Area Outline Pen und    */
SetAPen(Rast,2);
                 /*  Vordergrundfarbe.             */
Flood(Rast,0,440,50);
                 /*  Ellipse 2 im Modus 0 füllen   */

/*  Ein gefülltes Rechteck mit Farbe 1 und  Umran-
dung 2 Zeichnen */
SetAPen(Rast,1);
RectFill(Rast,100,110,540,170);

/*  TempRas und AreaInfo initialiseren   */
NewArea(Rast,4,&Remember);

/*   Ellipse 1 im Modus 1 füllen   */
Flood(Rast,1,120,50);

/*  Eine Gefüllte Ellipse mit farbe 2 Zeichnen.   */
SetAPen(Rast,2);
AreaEllipse(Rast,320,200,150,40);
Class = AreaEnd(Rast);

/* Auf Close Gadget warten und Fenster schließen   */
 Class = WaitEvent(Window,&code);
 /*  Auf  Close  Gadget warten     */
     CloseArea(Rast,Remember);
/*   Speicher   deallozieren       */
 CloseWindow(Window);
}
```

4. Gefüllte Polygone

Falls Sie sich die in AreaExtras definierte Prozedur ***PolyFill*** angesehen haben, werden Sie gemerkt haben, daß das Zeichnen von gefüllten Vielecken nicht ganz so einfach vor sich geht, wie es bei normalen Polygonen der Fall ist. Zunächst müssen, wie auch bei den Kreisen und Ellipsen, ***TempRas*** und ***AreaInfo*** entsprechend der Größe und Eckenzahl des Polygons initialisiert werden. Als nächstes muß mit ***AreaMove*** der Anfangspunkt des Polygons bestimmt werden. Nun können z.B. in einer ***for***-Schleife die erwünschten Eckkoordinaten mit ***AreaDraw*** dem System mitgeteilt werden, und schließlich das Polygon mit ***AreaEnd*** ausgegeben werden. An ***AreaMove*** und ***AreaDraw*** wird derZeiger auf den Rastport und das Koordinatenpaar übergeben, an ***AreaEnd*** nur der Zeiger auf den Rastport. So kann ein gefülltes Dreieck mit den Eckkoordinaten (100,10), (200,100), (0,200) nach diesem Verfahren, wie folgt gezeichnet werden.

Fig 5.1 Zeichnen eines gefüllten Polygons.

```
NewArea(Rast,3,&Remember);
AreaMove(Rast,100,10);

AreaDraw(Rast,100,10);
AreaDraw(Rast,200,100);
AreaDrwa(Rast,0,200);

AreaEnd(Rast);
```

Einfachheitshalber haben wir hier zur Initialisierung des *TempRas* und *AreaInfo* die Prozedur ***NewArea*** benutzt. Wesentlich einfacher können Sie ein gefülltes Vieleck unter Anwendung der schon erwähnten *PolyFill*-Prozedur erzeugen, die wir für Sie in AreaExtras vereinbart haben. Diese übernimmt nicht nur die *AreaMove*, *AreaDraw* und *AreaEnd* Aufrufe, sondern auch die Allozierung und Deallozierung des *TempRas* und des *AreaInfo.*

Als Eingabe braucht *PolyFill* wie auch die bekannte Systemroutine *PolyDraw* nur die Anzahl der Ecken, und die Adressen des Rastports und eines Arrays, in dem die Koordinaten der Ecken gespeichert sind. Um unser Dreieck auf diese Weise zu zeichnen, müssen Sie also folgendes tun:

(1) Ein Koordinatenarray so vereinbaren:

short Dreieck [] = {100,10, 200,100, 0,200};

(2) <u>PolyFill</u> so aufrufen:

PolyFill(Rast,&Dreieck[0],3);

Ein weiteres Beispiel für die Anwendug dieser Routine finden Sie im letzen Beispielprogramm dieses Kapitels.

5 Flächenmuster

Eine Fläche kann nicht nur mit einer Farbe, sondern auch mit einem Muster gefüllt werden. Das Muster muß eine Breite von 16 Punkten und die Höhe einer Zweierpotenz also 1,2,4,8....usw. haben. Die Bilddaten für das Muster werden, wie auch bei einem Image (siehe Kapitel 3) der Tiefe 1, in einem Array von ***USHORT*** Zahlen gespeichert, von denen jeder einer Bildzeile entspricht. In diesen Zahlen ist dann jedes Bit stellvertretend für einen Punkt. Ist das Bit gesetzt, so erscheint dieser Punkt in dem Muster, ist es gelöscht, dann erscheint er nicht. Ein horizontales Streifenmuster, das abwechselnd aus einer Zeile gesetzter und nicht gesetzter Punkte besteht, hat somit diese Form:

1111111111111111
0000000000000000

und muß wie folgt definiert werden:

```
USHORT Muster [] = {0xffff,
                    0x0000
                   };
```

Ein so erzeugtes Muster können Sie mittels ***SetAfPen*** einem Rastport zuordnen. Dazu müssen Sie diese Routine wie folgt Aufrufen:

```
SetAfPen(Rast,&Muster,Rows);
```

Hier bedeutet ***Rast*** einen Zeiger auf einen Rastport, ***Muster*** das Array, in dem die Bilddaten gespeichert sind, und ***Rows*** eine Zweierpotenz, die die Anzahl der Zeilen des Musters bestimmt (in unserem Beispiel müßte hier eine 1 stehen, denn 2 hoch 1 = 2). Nach diesem Aufruf wird beim Füllen bis auf weiteres dieses Muster verwendet, d.h. es werden nur die in dem Muster gesetzten Punkte in die Fläche hineingezeichnet (Zumindest wenn Sie als Zeichenmodus JAM1 benutzen). Wichtig ist, daß bei der Verwendung von Mustern alle Füllroutinen, also auch ***RectFill*** und ***Flood*** auch im outline-Modus einen ***TempRas*** benötigen!

Um das Füllmuster auszuschalten, rufen Sie ***SetAfPen*** mit ***NULL*** als Zeiger auf Bilddaten und 00 als Anzahl der Zeilen auf.

Ein Beispiel für die Verwendung von Füllmustern finden Sie in dem Programm Fill am Ende dieses Kapitels.

6 Farbmuster

Die Farbmuster sind eine Variante der normalen Füllmuster, die es möglich macht, Flächen auch mehrfarbig zu füllen. Dabei geben Sie für jede Bitplane ihrer Anzeige ein eigenes Muster an. Beim Füllen wird dann jede Bitplane mit dem zu Ihr gehörenden Muster gefüllt. Dadurch können Sie für jeden Punkt ihres Musters eine eigene Farbe angeben. Zur Verdeutlichung stellen Sie sich einmal vor, daß Sie für einen aus zwei Bitplanes bestehenden Screen folgendes Muster definieren:

Bitplane 0

```
1111111111111111
1111111111111111
0000000000000000
0000000000000000
```

Bitplane 1

```
0000000000000000
1111111111111111
1111111111111111
0000000000000000
```

Da die erste Zeile des Musters nur in der Bitplane Einsen hat, werden alle Punkte in dieser Zeile die Farbe binär 01 = dezimal 1 haben. In der zweiten Zeile gibt es sowohl in der Bitplane 0 als auch in der Bitplane 1 Einsen, so daß diese Zeile die Farbe binär 11 = dezimal 3 haben wird. Analog werden die vorletzte Zeile, wo sich alle Einsen in der Bitplane 1 befinden die Farbe binär 10 = dezimal 2, und die letzte, die in beiden Bitplanes nur Nullen hat, die Farbe 0 haben. Man könnte natürlich auch durch entsprechendes Abwechseln von Nullen und Einsen alle vier Farben innerhalb einer Zeile mischen. Die Bilddaten des Farbmusters werden, wie es auch bei den normalen Mustern der Fall war, in einem ***USHORT*** Array gespeichert, wobei die Daten für die einzelnen Bitplanes einfach hintereinander angeordnet sind.

Die Bildaten zu dem soebenen besprochenen Farbmuster sehen demnach folgendermaßen aus:

```
USHORT Farbmuster [] = {0xffff,   /* Bitplane 0  */
                         0xffff,
                         0x0000,
                         0x0000,

                         0x0000,   /* Bitplane 1  */
                         0xffff,
                         0xffff,
                         0x0000
                        }
```

Auch die Zuordnung des Muster an einen Rastport sieht hier fast genauso wie bei den Füllmustern aus. Der einzige Unterschied besteht darin, daß die Zweierpotenz, die die Anzahl der Zeilen bestimmt, als eine negative Zahl angegeben wird. Um unser Beispielmuster einem Rastport mit der Adresse ***Rast*** zuzuordnen, müßte man also folgendermaßen vorgehen:

```
SetAfPen(Rast,&Farbmuster,-2);
```

Zum Abschluß des Kapitels noch ein Beipielprogramm, daß die Anwendung der ***PolyFill***-Routine und der Füll- und Farbmuster demonstriert. Es definiert ein Füll- und ein Farbmuster, und zeichnet dann mit jedem eine gefüllte Raute.

Programm 5.4 Muster

```
#include "Display.h"
#include "AreaExtras.h"
#include "graphics/gfxmacros.h"

struct Window   *Window;
struct RastPort *Rast;
```

```
short  Poly1 [] = {320, 15,
                   630, 125,
                   320, 245,
                   10,  125
                  };

short  Poly2 [] = {320, 50,
                   520, 125,
                   320, 205,
                   120,  125
                  };

USHORT  Pattern [] = {0xffff,
                      0xf00f,
                      0xf00f,
                      0xffff
                     };

USHORT  ColPattern [] = { 0xffff,
                          0xf00f,
                          0xf00f,
                          0xffff,
                          0xffff,
                          0x0ff0,
                          0x0ff0,
                          0xffff
                        };

main ()
{

 USHORT code;
 ULONG  Class;

 OpenIntui();
 OpenGfx ();
```

```
/*  Fenster mit Close- und Depth-Gadget und  IDCMP-
Port öffnen.  */
Window = (struct Window *)MakeWindow(0,0,640,250,0,
                              0,"Fill",SMART_REFRESH|
WINDOWCLOSE|WINDOWDEPTH,CLOSEWINDOW,NULL);
 if(Window == NULL)          /*  Fehler beim Öffnen.  */
   exit(FALSE);
Rast = Window->RPort;    /* Adresse des Rastports. */

/*  Flähenmuster zum Füllen einstellen,   */
SetAfPt(Rast,&Pattern[0],2);

/*  und ein gefülltes Polygon zeichnen.   */
PolyFill(Rast,&Poly1[0],4);

/*  Farbmuster zum Füllen einstellen,   */
SetAfPt(Rast,&ColPattern[0],-2);

/*  und ein gefülltes Polygon zeichnen.   */
PolyFill(Rast,&Poly2[0],4);

/* Auf Close-Gadget warten und Fenster schließen. */
     Class = WaitEvent(Window,&code);

/* Auf Close Gadget warten  */
    CloseWindow(Window);
}
```

Kapitel 6

======================================
Nützliche Prozeduren und Tricks
======================================

Bis auf die Textausgabe und die Handhabung der Fonts, sollten Sie die Grundlagen der Grafikprogrammierung auf dem Amiga beherrschen, bevor Sie in diesem Kapitel weiter vordringen. In diesem Kapitel werden wir noch ein paar Dinge erwähnen, die in keines der vorangegegangenen Kapitel so richtig passen, obwohl diese für die Grundlegende Grafikprogrammierung durchaus nützlich sind. Auf die etwas komplexeren Themen wie Blitter, eigene Viewports etc. werden wir uns dann im nächsten Kapitel stürzen. Einige dieser Themen, wie z.B. die ***Forbid*** und ***Permit*** Routinen, haben zwar vordergründlich nichts mit der Grafik-Library zu tun, der erfahrene Programmierer weiß aber, daß er sie früher oder später auch bei der Grafikprogrammierung benötigen wird.

1. Die GfxBase- und IntuitionBase-Strukturen

Diese beiden globalen Strukturen werden beim Öffnen der entsprechenden Library initialisiert, wobei die ***OpenLibrary*** Prozedur jeweils die Adresse als Ergebnis liefert. Falls Sie zum Öffnen der Library die in Kapitel 2 vorgestellte Prozedur ***OpenIntui*** bzw. ***OpenGfx*** benutzen, dann steht Ihnen diese Adresse in der globalen ***GfxBase-*** bzw. ***IntuitionBase***-Variable direkt zur Verfügung. In der ***GfxBase***-Struktur finden Sie unter anderem die Adresse des aktiven Viewports (im Feld ActiveView-ViewPort), die Liste der verfügbaren Fonts (in ***TextFonts***), die Interrupts für Blitter und Timer (in ***Timsrv*** bzw. ***Bltsrv***) und die aktuellen Display-Flags (in ***DisplayFlags***).

Um beispielsweise die Adresse des momentan aktiven Viewports in View einzulesen, müssen Sie also folgendermaßen vorgehen:

```
View = GfxBase->ActiveView->ViewPort;
```

Die *IntuitionBase*-Struktur enthält noch um einiges mehr an wichtigen Informationen. Dazu gehören z.B.die Adressen des aktiven Screens und des aktiven Fensters (in ***ActiveScreen*** bzw.***ActiveWindow***),ein Zeiger auf die ***Preferences***-Struktur (in Preferences), die Adressen der ***Image***-Strukturen einiger Systemimages (in ***Check-Image*** bzw. ***AmigaIcon***) und die Zeiger auf die Systemgadgets (in ***SysGadgets***).

Besonderes nützlich ist auch die Adresse des ersten Screens der Intuition Screenliste, die Sie in ***First-Screen*** finden. Durch untersuchen des ***NextScreen***-Feldes dieser ***Screen***-Struktur können Sie die Adresse des nächsten ermitteln, und von dort auf die gleiche Weise die des übernächsten usw., so daß Sie die Möglichkeit haben, auf alle Screens zuzugreifen.

In der ***Screen***-Struktur befindet sich übrigens ein Feld namens ***FirstWindow***, das einen Zeiger auf das erste Fenster dieses Screens beinhaltet. Da jedes Fenster in ***NextWindow*** die Adresse des nächsten beinhaltet, stehen Ihnen so auch alle Fenster zur Manipulation offen!

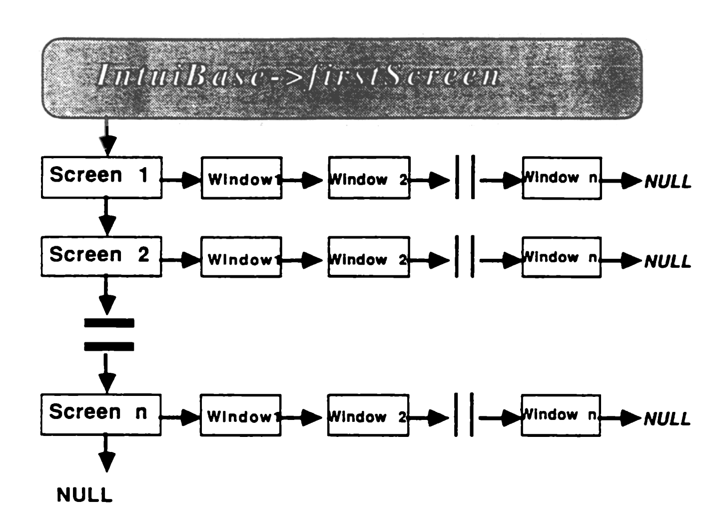


Bild 6.1 - Die Verknüpfung von Screens und Windows

Eine vollständige Behandlung der ***GfxBase***-und ***IntuitionBase***-Strukturen ist hier weder möglich noch notwendig.

Strukturen in den entprechenden include-Files (graphics/gfxbase., bzw. intuition/intuitionbase.) genauer ansehen. Wichtig ist, daß alle Felder zwar bedenkenlos gelesen werden können, jedoch nur mit äußerster Vorsicht verändert werden dürfen.

Ein Beispiel für einen sinnvollen Gebrauch der ***IntuitionBase***-Struktur finden Sie im nächsten Beispielprogramm.

2 Scrollen eines Rastports

Scrollen bedeutet nichts anderes, als ein verschieben von Auschnitten eines Bildes, und ist jedem von Ihnen von Spielprogrammen sicherlich bekannt. Die Graphics-Library enthält eine Prozedur, mit deren Hilfe ein Rechteck innerhalb eines Rastports um einen beliebigen Betrag in x- oder y-Richtung gescrollt werden kann.

Es handelt sich dabei um die Routine ScrollRaster, die als Eingabe die Rastportadresse (Rast), den zu scrollenden x- und y-Betrag (dx,dy) in Pixel und die Position und Größe des Rechtecks braucht (x1, y1 für die rechte obere Ecke, x2, y2 für die linke untere Ecke). Der Aufruf sieht dann so aus:

```
ScrollRaster(Rast,dx,dy,x1,y1,x2,y2);
```

Um also ein Quadrat mit der Seitenlänge 50 Pixel, dessen linke obere Ecke bei (10,20) liegt,um 30 Pixel nach rechts und 15 nach unten zu Scrollen, müssen Sie ***ScrollRast*** wie folgt benutzen:

```
ScrollRast(Rast,-30,-15,10,20,60,70);
```

Nach diesem Aufruf wird der gewünschte Bereich blitzschnell um den angegebenen Betrag bewegt und der freigewordene Platz mit der aktuellen Hintergrundfarbe gefüllt. Um den oft erwünschten Effekt des langsamen Scrollens zu erreichen, müssen Sie statt einmal

um einen großen Betrag, mehrmals um einen kleinen Scrollen, wobei gegebenfalls zwischen den einzelnen Scrollschritten eine Warteschleife anzubringen ist. Eine weitere oft erwünschte Variante des Scrollens ist das Rollen, d.h. daß das, was auf der einen Seite weggescrollt wird, auf der anderen wieder erscheint. Dies kann einfach dadurch erreicht werden, daß man den Bereich der "verschwinden" wird kopiert, bevor man anfängt zu Scrollen, und nach dem Scrollen auf der anderen Seite wieder einsetzt. Obwohl Kopierroutinen zu dem Thema Blitter gehören, das erst später behandelt wird, wollen wir hier wieder mal ein bißchen vorgreifen und eine solche Prozedur vorstellen. Sie heißt ***ClipBlit*** und erlaubt das Kopieren von Ausschnitten eines Rastports.

Das Aufrufformat hat die folgende Form:

```
ClipBlit(From,x1,y1,To,x2,y2,w,h,mask);
```

From und ***To*** sind hier die Zeiger auf den Quell-bzw. Zielrastport, ***x1*** und ***y1*** geben die Koordinaten an, an den sich die linke obere Ecke des zu kopierenden Bereiches befindet. Die Stelle, an die der Ausschnitt im ***To*** Rastport kopiert wird, ist durch ***x2*** und ***y2*** (wieder die Koordinaten der linken oberen Ecke), seine Größe durch ***w*** und ***h*** (Breite und Höhe), bestimmt. Die Bedeutung des letzten Paramter werden wir später im Zusammnenhang mit weiteren Blitterroutinen kennenlernen. Falls Sie mit dieser Routine nicht zwischen zwei verschiedenen Rastports, sondern innerhalb eines und desselben kopieren wollen, dann weisen Sie einfach ***From*** und ***To*** den gleichen Wert zu. Genau dies tun wir auch in dem hier abgedruckten Beispielprogramm, mit dem sich der gesamte momentan aktiver Screen einmal "herumrollt". Dazu wird 639 mal die

vorletzte Spalte der Anzeige in die erste kopiert und dann der ganze Rastport des Screens um ein Pixel nach rechts gescrollt. Dabei geht leider das, was sich vor dem Starten des Programms in der ersten Spalte befand, verloren, wir kommen dafür aber ohne einen zusätzlichen Rastport als Zwischenspeicher aus (Das korrekte Verfahren sähe so aus: die vorletzte Spalte vor jedem Scrollschritt in einen anderen Rastport kopieren, und erst danach in die erste Spalte einsetzten!).

Um die Adresse des Rastports des aktiven Screens zu ermitteln, wird wie im vorigen Abschnitt beschrieben, auf die *IntuitionBase*-Struktur zugegeriffen.

Programm 6.1 ScreenWrap

```
#include "exec/types.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "graphics/gfx.h"
#include "Display.h"
#include "stdio.h"

struct Screen      *Screen;
struct RastPort    *Rast;

main ()
{

 SHORT i;

 OpenIntui ();
 OpenGfx ();
```

```
/* Adresse des aktiven Screens und dessen Rastports
                                        ermitteln */
Screen = IntuitionBase->ActiveScreen;
Rast   = &((*Screen).RastPort);

/* Und den aktiven Screen ein mal herumrscrollen */
for(i = 1; i < 640; i++)
{
 ClipBlit(Rast,639,1,Rast,1,1,1,255,0x00c0);
 /* Eine Spalte kopieren */
 ScrollRaster(Rast,-1,0,1,1,639,255);
 /* Um eine Spalte Scrollen */
}
}
```

3 Definition eines eigenen Mauszeigers

Oft, wie z.B. bei Malprogrammen, wird der Benutzer Ihres Programms irgend etwas mit Hilfe des Mauszeigers eingeben müssen. Dabei wäre es in vielen Fällen schön, wenn der Mauspfeil eine Form hätte, die dem Zweck der Eingabe verdeutlicht, er also etwa wie ein Zeichenstift oder Pinsel aussehen würde. Der Mauszeiger darf, ähnlich wie ein Image (siehe Kapitel 3) beliebig hoch, aber nur 16 Pixel breit sein. Da der Zeiger ein Sprite ist, besteht er immer aus zwei Bitplanes, also bis zu vier Farben. Wichtig ist, daß die Farbe Null hier nicht dem Farbregister Null des Screens, sondern dem Screenfarbregister 32 enspricht (dies ist das Spritefarbregister Null). Ein wesentlicher Unterschied zur Definition von Imagedaten besteht auch darin, daß die Daten der beiden Bitplanes nicht in Blöcken hintereinander, sondern abwechselnd

eine Zeile (als ein Wort= eine USHORT Zahl) der einen und der anderen, gespeichert werden. Ein zwei Zeilen hoher Zeiger, der in der Bitplane 0 nur Einsen (gesetzte Punkte) und in der Biplane 1 nur Nullen (nicht gesetzte Punkte), also die Spritefarbe 1 hat, müßte demnach solche Bilddaten haben:

Abb 6.1 Beispiel für Mauszeigerdaten

```
INT data[] = {0xffff,   /* Bitplane 0 Zeile 1  */
              0x0000,   /* Bitplane 1 Zeile 1  */
              0xffff,   /* Bitplane 0 Zeile 2  */
              0x0000    /* Bitplane 1 Zeile 2  */
             };
```

Wie auch bei einem Image, müssen sich diese Daten im CHIP-RAM befinden. Wie Sie den notwendigen CHIP-Speicher allozieren, und die Daten dorthin kopieren können, haben wir bereits in Kapitel 3 gezeigt.

Wenn Sie die Daten von unserem Beispiel in einen CHIP-RAM Bereich kopiert haben, dessen Adresse in Data steht, dann durch diesen ***SetPointer*** Aufruf:

SetPointer(Window,Data,16,2,8,1);

der neue Zeiger wird dem durch Window gegebenen Fenster zugewiesen (Window ist ein Zeiger auf die ***Window***-Struktur). Um die Zuweisung wieder rückgängig zu machen, also um den normalen Mauspfeil wieder zu bekommen, können Sie ***ClearPointer*** mit dem Zeiger auf das Fenster so benutzen:

ClearPointer(Window);

Wie die Erzeugung eines eigenen Mauszeigers in der Praxis aussieht, können Sie auch an Hand des nachfolgenden Beispielprogramms sehen. In diesem Programm wird auch eine weitere "lustige" Routine verwendet. Gemeint ist die Prozedur ***DisplayBeep***, die Aufblitzen des gesamten Screens verursacht. Sie wird z.B. dazu verwendet die Aufmerksamkeit des Benutzer auf das Auftreten eines Fehlers, oder eines anderen wichtigen Ereignisses zu lenken. Falls Sie der Routine als Screenzeiger NULL übergeben, werden alle Screens aufblitzen.

Programm 6.2 Pointer.c

```
#include "intuition/intuition.h"
#include "exec/memory.h"
struct Window   *Window;

USHORT  Pointer[] = {0x0000,0x0000,
             0x600c,0x8001,    /* 2011000000001102 */
             0x0c30,0x8001,    /* 2000110000110002 */
             0x03c0,0x8001,    /* 2000001111000002 */
             0x03c0,0x8001,    /* 2000001111000002 */
             0x0c30,0x8001,    /* 2000110000110002 */
             0x600c,0x8001,    /* 2011000000001102 */
             0xc003,0xffff,    /* 3322222222222233 */
                    };


main ()
{

 SHORT  i;
 USHORT code;
 ULONG  Class;
 APTR   PData;
```

```
OpenIntui();
/*  Fenster mit Close- und Depth Gadget und IDCMP-
                                   Port öffnen.   */
Window = (struct Window *)MakeWindow(0,0,640,100,
                      0,0,"Pointer",SMART_REFRESH|
                    WINDOWCLOSE,CLOSEWINDOW,NULL);
if(Window == NULL)    /*  Fehler beim Öffnen.     */
  exit(FALSE);

/*  CHIP-Memory für Pointerdaten allozieren   */
PData = AllocMem(sizeof(Pointer),
                           MEMF_CHIP | MEMF_PUBLIC);
if (PData == NULL)    /*  Fehler beim Allozieren  */
 exit(FALSE);

/*  und Daten dorthin kopieren.  */
CopyMem(&Pointer[0],PData,sizeof(Pointer));

/*  Und den Pointer dem Fenster zuweisen   */
SetPointer(Window,PData,8,16,-8,-4);

/*  Screen blinken lassen   */
Class = WaitEvent(Window,&code);
                    /* Auf Close Gadget Warten.   */

/*  Speicher für Pointerdaten wieder freigeben  und
                                alles schließen.  */

CloseWindow(Window);
FreeMem(PData,sizeof(Pointer));

}
```

4 Beeinflussung des Multitaskings

Wie jeder Amiga-Benutzer wahrscheinlich weiß, kann die Darstellung von komplexen Grafiken auch am Amiga sehr lange dauern. Es ist daher oft erwünscht, für das eigene Programm, das eine solche Grafik erzeugt, möglichst viel Rechenzeit zu bekommen. Dies bedeutet, daß man diese Zeit anderen parallel laufenden Programmen "klaut". Wir wollen hier ein paar Routinen vorstellen, mit deren Hilfe es möglich ist, den Anteil der Rechenzeit, die Ihrem Programm zur Verfügung steht, zu vergrößern, bzw. im Extremfall alles andere (inklusive Maus und Tastaturabfrage !) abzuschalten, und somit die gesamte beträchtliche Rechenleistung des Amiga's im Dienste ihres Programms zu stellen. An dieser Stelle noch eine Warnung: Das Multitasking ist ein sehr empfindliches System, daß bei unüberlegten Manipulationen sehr schnell und oft auf sehr merkwürdige Art und Weise abstürzt (so daß nicht mal ein Reset hilft, und nur der Griff zum Powerschalter übrigbleibt).

Die mildeste Art den Rechenzeitanteil eines Programms der Prozedur ***SetTaskPri*** mit dem Zeiger auf die Task (Task) und einer Zahl zwischen -127 und 127, die die gewünschte Priorität angibt (Pri), so aufrufen:

SetTaskPri(Task,Pri);

Der Zeiger auf den eigenen Task kann so mit der ***FindTask***-Routine ermittelt werden:

Task = FindTask(NULL);

Eine normale Task hat eine Priorität von -20, sehr wichtige Systemtasks von bis zu +20. Sollten Sie die Priorität ihres Programms viel größer als die +20 machen, dann können Sie gleich alle anderen Tasks abschalten, weil sowieso nichts mehr läuft, sollten Sie sie dafür viel kleiner als die -20 machen, dann können Sie Ihr Programm genausogut abschalten, weil es sowieso keine Rechenzeit mehr abbekommt.

Wenn Sie sich dafür entschieden haben, alle anderen Tasks abzuschalten, dann können Sie die parameterlose ***Forbid***-Prozedur aufrufen. Mit Hilfe der ebenso parameterlosen ***Permit***- Routine können Sie diese wieder einschalten. Beachten Sie, daß es sehr unhöflich und darüberhinaus ein sehr schlechter Programierstil ist, über eine längere Zeit alle anderen Programme einfach auszuschalten!

Alle hier besprochenen Prozeduren sind in exec/tasks. zu finden.

Kapitel 7

====================
Textdarstellung
====================

Auf Grund der Tatsache, daß der Amiga keinen getrennten Textbildschirm besitzt, sondern den Text als Grafik darstellt, kann die Textausgabe extrem flexibel gestaltet werden. So kann der Text nicht nur Zeilen- bzw. Spaltenweise, sondern Pixelweise positioniert werden. Auch die Anzahl der möglichen Schriftarten ist praktisch unbegrenzt. Diese können sich nicht nur in der Form, sondern auch in der Höhe, Breite, sowie im Zeichenabstand unterscheiden. Wie es aber bei leistungsfähigen Systemen ist, ist es leider nicht ganz einfach, die angebotenen Möglichkeiten auszuschöpfen. Aus diesem Grund ist auch dieses Kapitel ein bißchen länger geraten. Wir versuchen hier, Ihnen an Hand einfacher Beispiele zu zeigen, was und wie es auf dem Amiga möglich ist.

1. Die Text-Prozedur

Als erstes wollen wir uns mit der Text-Routine befassen, die die Ausgabe einer beliebigen Zeichenkette in einem Rastport ermöglicht. Beim Aufruf wird ihr ein Zeiger auf den Rastport in dem der Text erscheinen soll, die Adresse der Zeichenkette und die Anzahl der auszugebenen Zeichen übergeben. Sie können also z.B. den Text "Hallo" wie folgt in den durch Rast bestimmten Rastport ausgeben:

```
Text(Rast,"Hallo",5);
```

Um nur den Text "Hal", also die drei ersten Zeichen der Zeichenkette auszugeben, müßten Sie eine 3 statt der 5 übergeben. Vieleicht haben Sie sich gewundert, daß keine Bildschirmposition für den Text angegeben wurde. Dies liegt daran, daß die Text-Prozedur den Text an der aktuellen Position des Grafikcursors in dem betroffenen Rastport ausgibt. Um den Text zu positionieren, müssen Sie also vorher die schon bekannte Prozedur Move aufrufen. Die Ausgabe der in String gespeicherten Zeichenkette an der Bildschirmposition (x,y) sieht also so aus:

```
Move(Rast,x,y);
Text(Rast,&String,Len);
```

Auch die Farben, Zeichenmodus und Schriftart (nicht Font) können durch den Aufruf entsprechender Prozeduren für die Textausgabe voreingestellt werden. Die Prozeduren ***SetAPen***, ***SetBPen***, und ***SetDrMd***, die zur Einstellung der ersten beiden Eigenschaften dienen, sind Ihnen ja schon bekannt (siehe Kapitel 4). In dem nächsten Beispielprogramm werden Sie die für die Textdarstellung interessanten Zeichenmodi finden. Dabei handelt es sich um die Modi ***JAM1***, ***JAM2***, ***INVERSVID*** und deren Kombinationen. Die Schriftart kann mit Hilfe von ***SetSoftStyle*** bestimmt werden. Diese ermöglicht Ihnen die Wahl einer Schrift die kursiv, fett, unterstrichen oder alles zu gleich ist.

Der Aufruf der Routine sieht wie folgt aus:

```
OldStyle = SetSoftStyle(Rast,Styles,Enable);
```

Rast ist hier wie immer ein Zeiger auf einen Rastport. In Enable müssen Sie die Flags (siehe Tabelle), die den gewünschten Schriftarten entsprechen, überbergeben. Als Funktionsergebnis gibt Ihnen die Prozedur die der alten Schriftart entsprechenden Flags zurück. Die ***FontStyle***-Flags werden in ULONG-Variablen gespeichert.

Abb 7.1 Die FontStyle-Flags

Gesetzte Flags	Resultierende Schriftart
FSF_BOLD	Fettschrift
FSF_ITALIC	Kursivschrift
FSF_UNDERLINED	Unterstrichen
FS_NORMAL	Normale Schrift

Durch Verknüpfen der Flags mit oder (¦) können Sie die erwähnte Mischung verschiedener Schriftarten erreichen (also z.B. fett und unterstrichen durch setzen von Enable auf FSF_BOLD ¦ FSF_UNDERLINED).

In der Variable Styles müssen die Flags aller für den aktuellen Font noch verfügbaren Schriftarten stehen, bzw. 255 wenn alle Schriftarten verfügbar sind. Diese können Sie durch einen Aufruf der AskSoftStyle-Prozedur wie folgt ermitteln:

```
Fonts = AskSoftStyle(Rast);
```

Um eine eingestellte Schriftart zu verändern, müssen Sie zuerst mit FS_NORMAL die "normale" Schrift einschalten und dann durch einen wiederholten Aufruf von ***SetSoftStyle*** die neue setzen. Dies gilt allerdings nicht, wenn Sie zu der aktuellen Schriftart eine andere hinzufügen wollen (also z.B.aus Fettschrift fette Kursivschrift machen wollen).

In diesem Fall können ***SetSoftStyle*** direkt das entsprechende Flag übergeben. Zum besseren Verständnis des Umgangs mit den verschiedenen Schriftarten und Zeichenmodi folgt wieder ein Beispielprogramm. Es öffnet ein Fenster, ermittelt mit Hilfe von ***AskSoftStyle*** die verfügbaren Schriftarten und gibt Texte mit verschiedenen Zeichenmodi und Schriftarten in dem Fenster aus.

Programm 7.1 Text

```
#include "intuition/intuition.h"
#include "graphics/gfxmacros.h"
#include "graphics/gfx.h"

struct Window   *Window;
struct RastPort *Rast;

main ()
{

 SHORT  i;
 USHORT code;
 ULONG  Class, Styles, OldStyle;

 OpenIntui();
 /*  Fenster öffnen und die Adresse des Rastports
                                          lesen.   */
```

```
Window = (struct Window *)MakeWindow(120,40,400,
                       130,0,0,"Text",SMART_REFRESH|
                       WIDOWCLOSE,CLOSEWINDOW,NULL);
if(Window == NULL)
                      /*  Fehler beim Öffnen.     */
  exit(FALSE);
Rast = Window->RPort;

/*  Verfügbare Schriftarten ermitteln  */
Styles = AskSoftStyle(Rast);

SetAPen(Rast,1);

Move(Rast,2,15);
Text(Rast,"Normal",6);

SetBPen(Rast,2);
SetDrMd(Rast,JAM2);
Move(Rast,2,30);
Text(Rast,"JAM2-Modus",10);

SetDrMd(Rast,JAM2|INVERSVID);
Move(Rast,2,45);
Text(Rast,"JAM2/COMPLEMENT-Modus",21);

SetBPen(Rast,0);

SetDrMd(Rast,JAM1);
OldStyle = SetSoftStyle(Rast,Styles,FSF_ITALIC);
Move(Rast,2,60);
Text(Rast,"Kursivschrift",13);
OldStyle = SetSoftStyle(Rast,Styles,FSF_BOLD);
Move(Rast,2,75);
Text(Rast,"Fette Kursivschrift",19);

OldStyle = SetSoftStyle(Rast,Styles,FSF_UNDERLINED);
Move(Rast,2,90);
```

```
    OldStyle = SetSoftStyle(Rast,Styles-OldStyle,255);
    OldStyle = SetSoftStyle(Rast,Styles,FSF_BOLD);
    Move(Rast,2,105);
    Text(Rast,"Nur Fettschrift",15);

    OldStyle = SetSoftStyle(Rast,Styles-OldStyle,255);
    OldStyle = SetSoftStyle(Rast,Styles,FSF_UNDERLINED);
    Move(Rast,2,120);
    Text(Rast,"Nur unterstrichen",17);

    /* Auf Close Gadget warten und Fenster schließen.*/
    Class = WaitEvent(Window,&code);
    CloseWindow(Window);
}
```

2. Die Fonts des Amiga

Der Begriff Fonts müßte eigentlich jedem Amiga Besitzer bekannt sein. Im Gegensatz zu älteren Computern, die nur einen Zeichensatz besaßen, den sie bestenfalls in verschiedenen Variationen wie fett oder invers darstellen konnten, kann der Amiga beliebige frei definierbare Zeichensätze (Fonts) verwenden. Die von der normalen Workbench bekannte Schrift heißt Topaz und ist im im ROM des Computers gespeichert. Alle anderen Fonts müssen entweder durch ein Programm definiert, oder von Diskette geladen und dann dem System zur Verfügung gestellt werden. Es ist dabei durchaus möglich, gleichzeitig in verschiedenen Fenstern verschiedene Fonts zu benutzen. Das System hat eine Liste der verfügbaren Fonts, aus der Sie Ihre Möglichkeiten schöpfen können.

Wie dies im Detail bewerkstelligt werden muß, werden Sie in späteren Abschnitten erfahren. Hier wollen wir zunächst ein paar allgemeine Worte zu den Fonts sagen. Wie schon gesagt, ist ein Font ein vollständiger Zeichensatz. Dies heißt, daß er aus 1 bis 255 beliebig definierten Zeichen bestehen kann. Sie können sich also einen Font erzeugen, der nur aus Grafikzeichen oder aber auch aus russischen oder griechischen Buchstaben besteht. Zu der Fontdefinition gehört die Angabe der Zeichenanzahl, deren Höhe, Breite, verfügbaren Schriftarten sowie einiger anderer besonderer Eigenschaften. Das Aussehen der Zeichen wird ähnlich wie ein Image Pixelweise durch Setzen einzelner Bits bestimmt.

Die auf Diskette gespeicherten Fonts befinden sich in der fonts-Directory der Systemdiskette. Dort gehört zu jedem Font ein eigenes Verzeichnis, in dem mehrere, sich in der Größe unterscheidende Versionen des Fonts stehen können. Falls Sie ihre eigenen Fonts aus einer anderen Directory verwenden wollen, müssen Sie dies dem System in der startup-sequence oder später im CLI-Fenster wie folgt mitteilen:

assign FONTS: MyFonts

Statt ***MyFonts*** müssen Sie hier natürlich den Namen ihres Verzeichnisses angeben. Beachten Sie dabei, daß ab dann alle fonts in Ihrem Verzeichnis gesucht werden.

3. Auflisten der Fonts

Hier können Sie nun definitiv erkennen und sehen, ob ein bestimmter Font geladen werden kann. Sie können alle Fonts, die für das System verfügbar sind, mit der ***AvailFonts***-Prozedur der Diskfont-Library aufgelistet werden. So einfach sich dies anhört, es ist leider mit einigen Aufwand verbunden. Beim Aufruf erzeugt ***AvailFonts*** eine Liste von ***AvailFonts***-Strukturen, die von einer ***AvailFontsHeader***-Struktur eingeleitet wird. Die letztere beinhaltet nur die Anzahl der folgenden ***AvailFonts***-Datensätzen und sieht so aus.

Abb 7.2 Die AvaiFontsHeader-Struktur

```
struct AvailFontsHeader
{
UWORD afh_NumEntries;
/*  struct AvailFonts afh_AF []  */
};
```

Wie sie der Abbildung entnehmen können, liegen die AvailFonts-Strukturen, die die eigentliche Information über die Fonts enthalten, direkt nach AvailFonts Header. Jede dieser Strukturen steht für einen Verfügbaren Font und sieht folgendermaßen aus:

Abb 7.3 Die AvailFonts-Struktur

```
struct AvailFonts
{
UWORD af_Type;
struct TextAttr af_Attr;
};
```

In af_Type wird der Fonttyp durch eine der folgenden Flags angegeben:

Flag	Bedeutung
AFF_MEMORY	Font befindet sich im Speicher.
AFF_DISK	Font befindet sich auf Diskette.

Die Daten des Fonts, also sein Name, seine Größe etc. sind in dem Feld ***af_Attr*** in einer weiteren Datenstruktur zu finden. Es handelt sich dabei um die *TextAttr*-Struktur, die zu den wichtigsten Strukturen der Grafik-Library gehört.
Sie ist wie folgt aufgebaut:

Abb 7.4 Die TextAttr-Struktur

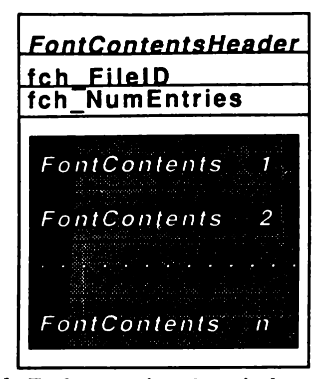
```
struct TextAttr
{
 APTR  ta_Name;
 UWORD ta_YSize;
 UBYTE ta_Style;
 UBYTE ta_Flags;
};
```

Den Zeiger auf den Namen des Fonts finden Sie in ***ta_Name***, die Höhe der Zeichen in ***ta_YSize***, und die Flags der verfügbaren Schriftarten (siehe Abschnitt 1) in ***ta_Style***. Das letze Feld, ***ta_Flags*** enthält ein oder meherere ***FontFlags***, die folgende Bedeutung haben.

Abb 7.5 Die Font-Flags

Flag	Bedeutung
FPF_ROMFONT	Font befindet sich im ROM
FPF_DISKFONT	Font ist auf Diskette
FPF_REVPATH	Font wird von links nach rechts geschrieben
FPF_TALLDOT	Font für non-interlace hires
FPF_WIDEOUT	Font für lowres/interlace
FPF_PROPORTIONAL	Font mit Proportionalschrift
FPF_DESIGNED	Fontbreite nicht einheitlich
FPF_REMOVED	Font wurde aus der Fontliste entfernt

Die Zusammenhänge zwischen den Strukturen verdeutlicht das Bild 7.1:



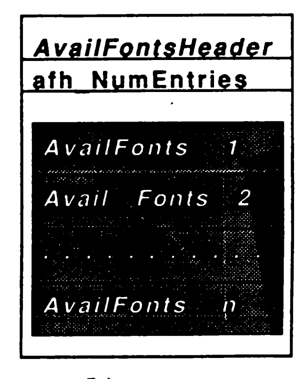


Bild 7.1 - Die Speicherung der Fontliste

Nach dieser relativ langweiligen Aufzählung der Datenstrukturen, können wir dazu übergehen, eine Fontliste zu erstellen. Als erstes muß für die Liste der *AvailFonts*-Strukturen genug Speicher alloziert werden. Dazu können Sie, in der aus den vorherigen Kapiteln bekannten Weise, die *AllocMem*-Routine von Exec verwenden. Der benötigte Speicher braucht nicht im CHIP-RAM zu liegen und sollte ca. 1000 Bytes groß sein. Sowohl den Zeiger auf den allozierten Speicher, als auch seine Länge, müssen Sie dann an AvailFonts übergeben. Als dritten Parameter müssen Sie die ange-angesiedelten Fonts, nur die Diskfonts oder beides auflisten wollen. Sollte es sich heraustellen, daß der von Ihnen allozierte Speicher nicht ausreicht, so bekommen sie als Funktionsergebnis von Typ long die Anzahl der fehlenden Bytes zurück. Sonst gibt die Prozedur 0 zurück. Das Erstellen einer Fontliste sieht also so aus:

```
Buffer = (struct AvailFontsHeader *)AllocMem
                                  (1000L,MEMF_PUBLIC);
e = AvailFonts(Buffer,1000L,Type);
```

Um nur die im Speicher befindlichen Fonts aufzulisten geben Sie für Type eine 2 an. Die Diskfonts bekommen Sie durch die Angabe einer 1, beides durch eine 3 aufgelistet.

Wie Sie die Daten aus der Fontliste herauslesen, ist der vorangegangenen Erklärung des Aufbaus dieser Liste leicht zu entnehmen. Da Sie in ***Buffer*** die ***Adresse*** des Headers ihrer Liste also der ***AvailFonts Header***-Struktur haben, können Sie die Anzahl der gelesenen Fonts leicht wie folgt ermitteln:

```
Number = Buffer>af_NumEntries;
```

Als nächstes müssen wir uns einen Zeiger auf die erste *AvailFonts*-Struktur beschaffen. Diese befindet sich aber an der Adresse von *AvailFontsHeader* plus der Länge des Feldes *af_NumEntries*.

Ihre Adresse kann also so ermittelt werden:

```
FirstFont = (struct AvailFonts *)&Buffer[1];
```

Nun können Sie in einer for-Schleife, in der Sie *FirstFont* inkrementieren und die *AvailFontsHeader-af_NumEntries*-mal durchlaufen wird, die Daten aller Fonts auslesen.

Wie dies im einzelnen gemacht wird, können Sie dem nachfolgenden Programm entnehmen. Beachten Sie, daß die Diskfont-Library geöffnet werden muß.

```
#include "libraries/diskfont.h"
#include "graphics/gfx.h"
#include "Display.h"
#include "stdio.h"
#include "exec/memory.h"

struct AvailFonts       *AFonts;
struct AvailFontsHeader *AFHeader;

ULONG  DosBase;
ULONG  DiskfontBase;
main ()
{
 LONG  e;
 SHORT i;
```

```
/*  Dos- und Diskfontlibrary öffnen.  */
OpenGfx();
if((DiskfontBase = OpenLibrary("diskfont.library"
                                        ,0)) == NULL)
 exit(FALSE);

if((DosBase = OpenLibrary("dos.library",0)) ==
                                              NULL)

/*  Speicher für die AvialFontsHeader-Struktur
                                        allozieren  */
AFHeader = (struct AvailFontsHeader *) AllocMem
                                    (5000,MEMF_CLEAR);
if (AFHeader == NULL)
 exit(FALSE);
          /*  Fehler beim Allozieren  */

/*  Liste der verfügbaren Memory- und Diskfonts
                                         einlesen.  */
e = AvailFonts(AFHeader,5000L,3);

/*  Die eingelesenen Fonts auflisten.  */
AFonts = (struct AvailFonts *)&AFHeader[1];
                       /*  Header überlesen.      */

for (i = 0; i < (AFHeader->afh_NumEntries); i++)
                       /* Alle Einträge listen.  */
{
 /*  Den Namen und die Größe des Fonts ausgeben  */
  printf("Font Nr %-2d  %-24s Größe %d \n",i,AFonts
        ->af_Attr.ta_Name,Afonts>af_Attr.ta_YSize);

 AFonts++;                    /*  Nächster Font.  */

 }
}
```

4. Öffnen eines Fonts

Hat man erst herausgefunden, welche Fonts einem eigentlich zur Verfügung stehen, wird man wohl auch zumindestens einen laden wollen. Dazu benutzt man die schon aus dem vorherigen Abschnitt bekannte *TextAttr*-Struktur. Dort trägt man den Namen, die Größe und die Art des Fonts (Disk oder Speicher) ein.

Die Adresse der so initialisierten Struktur übergibt man dann der ***OpenFont*** bzw. der ***OpenDiskFont***-Prozedur. Welche der beiden man nun benutzt hängt davon ab, ob Sie auch nicht vergessen, am Anfang des Programms die Diskfont-Library zu öffnen. Ganz egal welche der Prozeduren Sie verwendet haben, das Ergebnis bleibt gleich. Falls der gewünschte Font geladen werden konnte, liefern beide einen Zeiger auf die TextFont-Struktur des neuen Fonts (anderenfalls bekommen Sie NULL zurück). Diese Struktur, die Sie im nächsten Abschnitt genauer kennenlernen werden, liefert eine genaue Beschreibung des Fonts.

Sie beinhaltet alles, was die Systemsoftware braucht, um mit diesem Font zu arbeiten, wie z.B. den Zeiger auf die Bilddaten der einzelnen Zeichen. Nachdem der Font erfolgreich geöffnet wurde, haben Sie zwei Möglichkeiten mit ihm zu arbeiten. Die erste besteht darin, ihm mit Hilfe der ***SetFont***-Prozedur an einen Rastport (z.B. den eines Fensters oder eines Screens) zuzuweisen. Das Öffnen eines Fonts und die Zuweisung an einen Rastport Rast kann also aussehen, wie folgt:

```
TextFont = (struct TextFont *)OpenDiskFont(TextAttr);
if(TextFont != NULL) SetFont(Rast,TextFont);
```

Wir haben hier vorausgesetzt, daß ***TextAttr*** auf eine initialisierte ***TextAttr***-Struktur zeigt. Für alle Texte, die dann in diesem Rastport mit Text ausgegeben werden, wird nun der neue Font verwendet.

Ein Beispiel für das Öffnen eines Diskfonts und deren Zuweisung an einen Rastport finden sie in dem an diesem Abschnitt folgenden Programm. Es gibt einen Text mit dem Standardfont des Fensters aus, öffnet dann nacheinander zwei neue Fonts und benutzt diese ebenfalls zur Textausgabe. Die von dem Programm verwendetet Fonts: Emerald 17 und Emerald 20.

Wenn Sie ihren Amiga von einer Diskette gebootet haben, auf der diese Fonts nicht vorhanden sind, dann müssen Sie vor dem Programmstart die Workbench ins Laufwerk schieben und mit assign eine neue FONTS: device setzen (siehe Abschnitt 2).

Programm 7.3 Diskfont

```
#include "Display.h"
#include "intuition/intuition.h"
#include "graphics/gfx.h"
#include "libraries/Diskfont.h"
struct Window    *Window;
struct RastPort  *Rast;
struct Font      *DFont;
struct TextFont  *TFont;

struct TextAttr ITFont = {"Emerald.Font",
                           17,
                           0,
                           FPF_DISKFONT
                          };
```

```
ULONG  DiskfontBase;

main ()

{
 SHORT  i;
 USHORT code;
 ULONG  Class, Styles, OldStyle;
 OpenIntui();
 OpenGfx();

 /*  Diskfontlibrary öffnen   */
 DiskfontBase = OpenLibrary("diskfont.library",0);
 if(DiskfontBase == NULL)
                               /*  Fehler beim Öffnen   */
  exit(FALSE);

 /*  Fenster öffnen un die Adresse des Rastports
                                            lesen.   */
 Window = (struct Window *)MakeWindow(120,10,400,
                                    100,0,0,"DiskFonts",
             SMART_REFRESH|WINDOWCLOSE|WINDOWDEPTH,
                                      CLOSEWINDOW,NULL);

 if(Window == NULL)   /*  Fehler beim Öffnen.      */
   exit(FALSE);
 Rast = Window->RPort;

 Move(Rast,120,30);
                     /*  Text im Standardfont ausgeben   */
 Text(Rast,"Standardfont",12);

 TFont = OpenDiskFont(&ITFont);
                                /*  Emerald 17 öffnen   */
 if(TFont == 0L)
                                         /*  Fehler ?   */
  exit(FALSE);
 SetFont(Rast,TFont);
```

```
              /*  Nein -> dem Rastport zuweisen      */

Move(Rast,120,50);
              /*  und Text mit Emerald 17 printen.  */
Text(Rast,"Emerald 17\n",11);

ITFont.ta_YSize = 20;       /*  Emerald 20 öffnen   */
TFont = OpenDiskFont(&ITFont);
if(TFont == OL)                      /*  Fehler ?  */
 exit(FALSE);
SetFont(Rast,TFont);
              /*  Nein -> dem Rastport zuweisen      */

Move(Rast,120,75);
Text(Rast,"Emerald 20\n",11);
              /*  und Text mit Emerald 20 printen.  */

/* Auf Close Gadget warten und Fenster schließen.*/
CloseWindow(Window);
}
```

Bild 7.2 - Die Ausgabe des Programms DiskFont

Als nächstes sollte noch die zweite Möglichkeit, einen geöffneten Font zu nutzen, erläutert werden. Dazu wird der Font mit ***AddFont*** an die Fontliste des Systems angehängt und kann von anderen Programmen benutzt werden. Ist ***TextFont*** ein Zeiger auf die entsprechende ***TextFont***-Struktur, dann sieht dies folgendermaßen aus:

```
AddFont(TextFont);
```

Für ihr eigenes Programm ist das Einfügen des Fonts in die Systemliste immer dann wichtig, wenn Sie mit Prozeduren oder Datenstrukturen arbeiten, den eine *TextAttr*-Struktur statt einer *TextFont*-Struktur übergeben werden muß. Dies ist z.B. dann der Fall, wenn Sie beim Öffnen eines Screens oder Fenster einen eigenen Font als Standardfont haben wollen.

Sowohl die ***NewWindow*** als auch die ***NewScreen***-Struktur beinhalten ein Feld, in dem Sie die Adresse einer *TextAttr*-Struktur angeben können. Sie muß dann mit den gleichen Werten initialisiert sein, die Sie zum Öffnen des Fonts verwendet haben, damit Sie auch die gleiche Struktur verwenden. Auf ähnliche Weise kann der Font bei der Textausgabe mit ***IntuiText*** bestimmt werden, die wir noch im letzten Abschnitt dieses Kapitels besprechen.

Wem das Laden fertiger Fonts nicht genug ist, für den wollen wir jetzt beschreiben wie man sich einen eigenen definieren und in das System einfügen kann.

Die Grundlage dafür ist die bereits vorher angesprochene *TextFont*-Struktur. Diese sieht folgendermaßen aus:

Abb. 7.6 Die TextFont-Struktur

```
struct TextFont
{
 struct Message tf_Message;
 UWORD tf_YSize;
 UBYTE tf_Style;
 UBYTE tf_Flags;
 UWORD tf_XSize;
 UWORD tf_Baseline;
```

```
 UWORD tf_BoldSmear;
 UWORD tf_Accessors;
 UBYTE tf_LoChar;
 UBYTE tf_HiChar;
 APTR  tf_CharData;
 UWORD tf_Modulo;
 APTR  tf_CharLoc;
 APTR  tf_CharSpace;
 APTR  tf_CharKern;
};
```

In dem Feld ***tf_Message*** befindet sich eine gewöhnliche ***Message***-Struktur von Exec, die notwendig ist, damit der Font in die Fontliste eingefügt werden kann. Deswegen muß dieses Feld wie folgt initialisiert werden:

```
TextFont.tf_Message.mn_Node.ln_Type = NT_FONT;
TextFont.tf_Message.mn_Node.ln_Name = Name;
TextFont.tf_Message.mn_Length       = Len;
```

TextFont muß hier natürlich eine Variable vom Typ ***TextFont*** sein. ***Name*** ist ein Zeiger auf den Namen des Fonts und Len der Speicherbedarf aller zu diesem Font gehörenden Daten. Wie dieser berechnet wird, werden wir später erläutern. Die nächsten vier Felder ***tf_YSize***, ***tf_Style***, ***tf_Flags*** und ***tf_XSize*** bestimmen die Höhe, die verfügbaren Schriftarten, die Art und die Breite des Fonts. Die Flags (***tf_Flags***) und die Schriftarten entsprechen den von ***TextAttr*** bzw. ***SetSoftStyle*** bekannten (siehe Abb 7.5 bzw 7.1).

In ***tf_Baseline*** wird der Abstand der untersten Zeile des Textes von oben festgelegt. Wenn Sie z.B. unten Platz zum Unterstreichen lassen wollen, dann müssen Sie hier einen Wert, der kleiner als die Fonthöhe ist, angeben. Durch Angabe einer 1 in ***tf_Bold***

Smear können Sie die Erzeugung der Fettschrift durch "Verschmieren" zulassen. Das nächste Feld: ***tf_Accessors*** ist nur für das System wichtig und sollte mit 0 initialisiert werden. Der Rest der TextFont-Struktur bezieht sich direkt auf die Zeichendaten. Zunächst werden durch ***tf_LoChar*** und ***tf_HiChar*** die ASCII-Werte des ersten und des letzten Zeichens des Fonts angegeben. Die Werte können zwischen 0 und 255 liegen, wobei ***tf_LoChar*** natürlich kleiner als ***tf_HiChar*** sein muß. Die Adresse des Speicherbereiches, in dem die Bildaten der Zeichen stehen, befindet sich in ***tf_CharData***. Wie es bei Bilddaten üblich ist, wird jeder Punkt durch ein Bit und jede Bildzeile durch eine Bitreihe repräsentiert, so daß die Zeichendaten in einem Array vom Typ UWORD definiert werden können. Die Zeichen werden "nebeneinander" gespeichert, d.h auf die erste Zeile des ersten Zeichens folgt die erste des zweiten, darauf die erste des dritten etc. Danach kommen auf die gleiche Art angeordnet die zweiten, dritten, vierten usw. Zeilen. Wie Sie sehen, ist die Anzahl der Bytes, die übersprungen werden müssen, um von einer Zeile eines Zeichens zur nächsten zu kommen, von Zeichensatz zu Zeichensatz verschieden (Es liegen ja die Zeilendaten der anderen Zeichen dazwischen !). Aus diesem Grunde muß die Anzahl der zwischen zwei Zeilen eines Zeichens liegenden Bytes in ***tf_Modulo*** angegeben werden. Bei einem Font, der 10 Zeichen enthält, die alle 16 Punkte breit sind (also genau zwei Bytes belegen), wärend es logischerweise 20 - da dieser Wert für alle Zeilen gleich bleibt, müssen alle Zeilen eines Zeichens auch gleich breit sein.

Das soll aber natürlich nicht heißen, daß alle Zeichen quadratich sein müssen. Wo kein Punkt erscheinen soll, kann ja in den Daten eine Null gesetzt werden. Da die Zeichenbreite aber nicht auf ganzzahlige Vielfache von 8 oder 16 beschränkt ist, wird in der

Regel eine Zeile eines Zeichens nicht durch eine bestimmte Anzahl von Bytes oder Words dargestellt. Es ist auch nicht unbedingt notwendig, daß alle Zeichen die gleiche Breite haben. So können in zwei Bytes einer Zeile eines 10 und eines 6 Bits breiten Zeichens gespeichert werden. Damit der Amiga in diesem Durcheinander überhaupt zurecht kommt, müssen in einem zusätzlichen Speicherbereich nacheinander der Abstand des Anfangs der ersten Zeile von Anfang der Bildaten und die Breite der Daten einer Zeile in Bits für jedes Zeichen angegeben werden. Ein Zeiger auf diesen Speicherbereich ist in ***tf_CharLoc*** einzutragen.

Die Breite des Zeichens wird für jedes Zeichen in einem weiteren Array, dessen Adresse in ***tf_CharSpace*** steht, übergeben. Der aufmerksame Leser wird an dieser Stelle wahrscheinlich verwundert sein, denn nun haben wir ***tf_CharLoc*** und ***tf_CharSpace***)! Der Grund dafür ist darin zu suchen, daß es sich jedesmal um eine "andere" Breite handelt. In ***tf_XSize*** wurde angegeben, wie viel Platz jedes Zeichen unabhängig von seiner Breite bei einem nicht proportionalen Zeichensatz auf dem Bildschirm horizontal einimmt. Dieser Wert ist für alle Zeichen des Zeichensatzes gleich und wird deswegen global in der ***TextFont***-Struktur festgelegt. Die Breite, die in ***tf_CharSpace*** bestimmt wird, kann von Zeichen zu Zeichen verschieden sein. Diese Breite wird immer, also auch bei proportionalen Fonts, beibehalten. Sie gibt Ihnen die Möglichkeit jedem Zeichen so viel Platz zuzuordnen, wie Sie es für eine gute Lesbarkeit, oder schönes Aussehen des Zeichensatzes für angebracht halten. Die in ***tf_CharLoc*** stehende Breite, gibt schließlich die Breite der Daten jedes Zeichens an. Dies ist auch die maximale Anzahl der horizontal gesetzen Punkte eines Zeichens. Die Differenz aus dieser und der vorigen Breite wird als freier Raum rechts vom Zeichen dargestellt. Damit Sie den Rand auch Links lassen oder das Zeichen zentrie-

ren können, gibt es noch in ***TextFont*** das Feld ***tf_CharKern***. Dort steht ein Zeiger auf ein Array, das für jedes Zeichen den Abstand des ersten Datenbits einer Zeile vom Anfang der Zeile, also die Verschiebung der Daten nach rechts, sprich den linken Rand angibt.

Alles klar? Nicht? Also noch ein Beispiel: bei einem Zeichen mit einer Datenbreite von 8 Bits, und einer Zeichenbreite von 10 Bits müßte hier eine Eins stehen, damit es links und rechts den gleichen Rand hat.

Für die, die immer noch verwirrt sind, haben wir ein (hoffentlich) einfaches Beispiel: Dazu stellen Sie sich bitte zunächst einen Zeichensatz vor, der nur aus zwei Zeichen besteht, die je 3 Zeilen hoch sind. Die

Zeichen sollen so aussehen:

```
Zeichen 1           Zeichen 2

111111111             101
100000001             010
111111111             101
```

Die Datenbreite des ersten Zeichens ist 9, die des zweiten 3, die Gesamtbreite einer Zeile also 12 Bits. Für die Zeilendaten brauchen wir also je ein Word (2 Bytes). Die Daten sehen so aus:

```
ULONG  CharData [] = {0xffd0,
                       0x80a0,
                       0xffd0
                      };
```

Der ***tf_Modulo***-Wert in dem zugehörenden ***TextFont***-Datensatz mu' hier den Wert 2 haben. In ***tf_CharLoc*** muß die Adresse des folgenden Arrays stehen:

```
ULONG CharLoc []  = {0x0000,0x0009,
                     0x0009,0x0003
                    };
```

Falls Sie noch wollen, daß beim ersten Zeichen sowohl links als auch rechts zwei Pixel freigelassen werden, während das zweite Zeichen um 3 Pixel nach links verschoben wird, dann müssen ***tf_ChaSpace*** und ***tf_CharKern*** wie folgt aussehen:

```
UWORD CharSpace [] = {13,6}
```

Mit den so erzeugten Rändern sehen die beiden Zeichen nun so aus.

```
Zeichen 1            Zeichen 2

0011111111100          10100
0010000000100          01000
0011111111100          10100
```

Wenn Sie die Fontdaten Ihres Zeichensatzes wie beschrieben erzeugt und die ***TextFont***-Struktur initialisert haben, können Sie den Font auch sofort benutzen. Da Sie die Adresse der ***TextFont***-Struktur schon kennen (Sie haben sie ja immerhin erzeugt!), brauchen Sie weder die ***OpenFont***-noch die ***OpenDiskFont***- Routine aufzurufen. Es reicht wenn Sie den Zeiger auf Ihre ***TextFont***-Struktur an ***SetFont*** oder ***AddFont*** übergeben (siehe Abschnitt 3).

Wie die Konstruktion eines neuen Zeichensatzes im einzelnen verläuft, können Sie sich nochmals an Hand des nachfolgenden Programms klarmachen. Es definiert sich fünf Grafikzeichen, deren ASCII-Werte zwischen

65 und 70 liegen. Diese Zeichen werden also immer dann ausgegeben, wenn bei dem normalen Font die Buchstaben "A" bis "E" erscheinen würden. Wichtig ist, daß alle anderen Zeichen nicht etwa im alten Font erscheinen, sondern überhaupt nicht verfügbar sind.

Bei dem Versuch Sie auszugeben wird gar nichts, oder eventuell nur Müll erscheinen.

Programm 7.4 NewFont

```
#include "graphics/gfx.h"
#include "graphics/text.h"
#include "libraries/Diskfont.h"
#include "exec/memory.h"

struct Window    *Window;
struct RastPort  *Rast;
struct TextFont  *NewFont;

/*  Definition des neuen Fonts für folgender Grafik-
zeichen                    */
/*  mit den ASCII-Codes von 65 bis 70:              */

/* 1111111111111111 0000001111000000 1111111111111111
00000000 11001100  */
/* 1100000000000011 0000001111000000 0000000000000000
00000000 11001100  */
/* 1100000000000011 0000001111000000 1111111111111111
00000000 11001100  */
/* 1100000000000011 0000001111000000 0000000000000000
00000000 11001100  */
/* 1100000000000011 0000001111000000 1111111111111111
00000000 11001100  */
/* 1100000000000011 0000001111000000 0000000000000000
00000000 11001100  */
/* 1100000000000011 1111111111111111 1111111111111111
```

```
00000000 11001100  */
/* 1111111111111111 1111111111111111 0000000000000000
00000000 11001100  */

UWORD  FontData []  = {0xffff,0x03c0,0xffff,0x00cc,
                        0xc003,0x03c0,0x0000,0x00cc,
                        0xc003,0x03c0,0xffff,0x00cc,
                        0xc003,0x03c0,0x0000,0x00cc,
                        0xc003,0x03c0,0xffff,0x00cc,
                        0xc003,0x03c0,0x0000,0x00cc,
                        0xc003,0xffff,0xffff,0x00cc,
                        0xffff,0xffff,0x0000,0x00cc,
                       };

/*  Die Adressen der Zeichen im Zeichensatz  */
UWORD Loc [] = {0x0000,0x0010,0x0010,0x0010,0x0020,
                0x0010,0x0030,0x0008,0x0038,0x0008
               };

/*  Zeichenbreite  */
UWORD Kern[]  ={0,0,0,0,0};

ULONG  DiskfontBase;

main ()
{

 SHORT  i;
 USHORT code;
 ULONG  Class, Len;

 OpenIntui();
 OpenGfx();

 /*  Diskfontlibrary öffnen  */
 DiskfontBase = OpenLibrary("diskfont.library",0);
 if(DiskfontBase == NULL )  /* Fehler beim Öffnen */
  exit(FALSE);
```

```
/*  Fenster öffnen un die Adresse des Rastports
                                          lesen. */
Window = (struct Window *)MakeWindow(220,100,200,
                                  100,0,0,"NewFont",
              SMART_REFRESH|WINDOWCLOSE|WINDOWDEPTH,
                                  CLOSEWINDOW,NULL);

if(Window == NULL)        /*  Fehler beim Öffnen.  */
  exit(FALSE);

Rast = Window->RPort;

/*  Speicherplatz für die TextFont-Struktur des
                           neuen Fonts allozieren  */
NewFont = AllocMem(sizeof(struct TextFont),
                             MEMF_PUBLIC|MEMF_CLEAR);
 if(NewFont == NULL)
  exit(FALSE);

/*  Länge der Fontdaten berechnen  */
Len = sizeof(struct TextFont) + sizeof(FontData) +
                   sizeof(Loc)+ sizeof(Width) +
                   sizeof(Kern);

/*  Und diese initialisieren   */
NewFont->tf_Message.mn_Node.ln_Type = NT_FONT;

/*  Message Port   */
NewFont->tf_Message.mn_Node.ln_Name = "Demofont";
NewFont->tf_Message.mn_Length       = Len;

/*  Eigentlicher TextFont  */
NewFont->tf_YSize          = 8;

/*  Höhe des Fonts  */
NewFont->tf_Style          = NULL;
NewFont->tf_Flags          = FPF_DESIGNED;
NewFont->tf_XSize          = 16;
```

```
/*  Breite der Zeichen        */
NewFont->tf_Baseline     = 0;
NewFont->tf_BoldSmear    = 1;
NewFont->tf_Accessors    = NULL;
NewFont->tf_LoChar       = 65;
NewFont->tf_HiChar       = 70;
NewFont->tf_CharData     = &FontData[0];
NewFont->tf_Modulo       = 8;
NewFont->tf_CharLoc      = &Loc[0];
NewFont->tf_CharSpace    = &Width[0];
NewFont->tf_CharKern     = &Kern[0];

AddFont(NewFont);
     /* Font in die Systemliste einfügen  */

Move(Rast,50,20);
       /*  Text im Standardfont ausgeben  */

Text(Rast,"Standardfont:",13);
Move(Rast,60,40);
Text(Rast,"ADBDCDE",7);
Move(Rast,50,60);
Text(Rast,"Neuer Font:",11);

SetFont(Rast,NewFont);
     /*  Neuen  Font dem Rastport zuweisen  */

Move(Rast,50,75);
     /*  und den Text mit diesen  printen.  */
Text(Rast,"ADBDCDE",7);

/*  Auf Close Gadget warten und Fenster schließen.*/
Class = WaitEvent(Window,&code);
RemFont(NewFont);

}
```

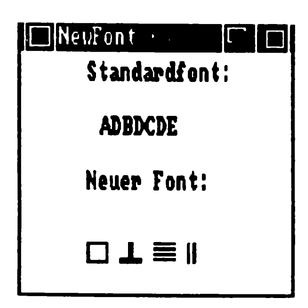


Bild 7.3 - Die Ausgabe des Programms NewFont

5. Erzeugen eines neuen Fonts auf Diskette

Wer einen eigenen Font erzeugen und auf Diskette speichern will, wird gewöhnlich zu einem der vielen Hilfsprogramme, mit deren Hilfe man die einzelnen Zeichen zeichnerisch erstellen kann, greifen.

Trotzdem mag es Sie vieleicht interessieren, wie dies genau funktioniert. Wenn ein Diskfont mit ***Open DiskFont*** geladen wird, dann wird, wie Sie sich vielleicht erinnern, ein ***TextFont***-Datensatz erzeugt, wie er im vorigen Abschnitt beschrieben wurde. Es ist daher durchaus nicht verwunderlich, daß die Fontdaten auf Diskette ebenfalls in einer sehr ähnlichen Form vorliegen. Der wesentliche Unterschied besteht indem, was sich um die eigentlichen Zeichendaten herum befindet. Wenn Sie einen Blick in die fonts-Directory der Workbenchdiskette werfen, dann werden Sie sehen, daß sich dort zu jedem Font zunächst mal ein gleichnamiges Verzeichnis und ein File befindet, deren Name aus dem Namen des Fonts und einem ".font"besteht. Für den aus Abschnitt 4 bekannten "Emerald"-Font sieht dies aus, wie folgt:

emerald (Dir)
emerald.font

In der ermerald Directory stehen dann nur folgende Files:

17
20

Was steckt dahinter? In dem ".font"-File befindet sich eine ***FontContentsHeader***-Struktur, der eine oder mehrere ***FontContents***-Strukturen folgen (ähnlich wie es auch bei ***AvailFontsHeader*** und ***AvailFonts*** der Fall war). ***FontContentsHeader*** gibt an, wieviele Fonts mit diesem Namen verfügbar sind und sieht so aus:

Abb 7.7 Die FontContentsHeader-Struktur

```
{
 UWORD fch_FileID;
 UWORD fch_NumEntries;
 /*  struct FontContents []  */
};
```

Das Feld ***fch_FileID*** kennzeichnet diese Sorte von Dateien und sollte immer 0f00 sein. In ***fch_NumEntries*** steht dann die Anzahl der nachfolgenden ***FileContents***-Strukturen. Jede dieser Strukturen ist stellvertretend für einen "Unterfont" (z.B. emerald 17 oder emerald 20) und ist wie folgt gebaut:

Abb 7.8 Die FontContents-Struktur

```
struct FontContents
{
char fc_FileName[MAXFONTPATH];
UWORD fc_YSize;
UBYTE fc_Style;
UBYTE fc_Flags;
};
```

Die *fc_YSize*, *fc_Style* und *fc_Flags* Felder beschreiben die Eigenschaften des Fonts und haben die gleiche Bedeutung wie die gleichnamigen Felder der *TextAttr*- und *TextFont*-Strukturen. *fc_FileName* gibt den Pfad, unter dem der Font in der FONTS: Directory zu finden ist. In unserem Beispiel mit dem emerald-Font muß hier ein Zeiger auf die Zeichenkette" emerald/17" oder "emerald/20" (Je nachdem, zu welchen der beiden Fonts diese Struktur gehört) stehen.

Die Datei, die den eigentlichen Zeichensatz beinhaltet, besteht dann aus folgenden vier Teilen:

(1) Eine einleitende ***DiskFontHeader***-Struktur.

(2) Die ***TextFont***-Struktur des Zeichensatzes.

(3) Die Bilddaten der Zeichen.

(4) Die ***CharLoc***, ***CharSpace*** und ***CharKern*** Daten (siehe ***TextFont***-Struktur).

Die Funktion der ***DiskFontHeader***-Struktur liegt darin, das Laden des Fonts mit dem ***LoadSeg***-Befehl zu ermöglichen. Dadurch kann ein Zeichensatzfile wie ein gelinktes Programm behandelt, und im Speicher installiert werden. Hier die ***DiskFontHeader***-Struktur im Detail:

Abb 7.9 Die DiskFontHeader-Struktur

```
struct DiskFontHeader
{
/*  ULONG  dfh_NextSegment  */
/*  ULONG  dfh_ReturnCode   */
```

```
struct Node dfh_DF;
UWORD dfh_Revision;
LONG dfh_Segment;
char dfh_Name[MAXFONTNAME];
struct TextFont dfh_TF;
};
```

Die ersten beiden Felder befinden sich in Kommentarklammern, weil sie zwar immer am Anfang der Struktur erscheinen, aber trotzdem nicht richtig dazu gehören.

In *dfh_ReturnCode* stehen für den Fall, daß jemand probieren sollte, das File als Programm zu starten, folgende Assemblerbefehle:

```
MOVEQ #0,D0
RTS
```

Die den Rücksprung zum aufrufenden Programm veranlassen und so einen Absturz verhindern.

In der *Node*-Struktur, die sich in *dfh_DF* befindet, sind nur die Felder *ln_Type* und *ln_Name* ungleich Null. Das Erste beinhaltet den Typ des Nodes, also *NT_FONT*, das Zweite einen Zeiger auf den Namen des Fonts. Die letzten drei Felder von *DiskFontHeader* haben immer die Werte DFH_ID, 1 und 0. Als nächstes folgen der Fontname (kein Zeiger auf den Namen!) in *dfh_Name* und die *TextFont*-Struktur.

Zur Erzeugung dieser Struktur, die in der im vorigen Abschnitt besprochene Art einen Zeichensatzes definiert, benutzen Sie am besten einen Assembler. Die Definition sieht dann schematisch so aus:

```
INCLUDE "exec/types.i"
INCLUDE "exec/nodes.i"

*Definition der DiskFontHeader-Struktur

MOVEQ #0,DO
RTS
DC.L                        0
.............................
DC.L                 fontName
.............................
fontName:
DS.B    MAXFONTNAME

*Definition der TextFont-Struktur

font:
DC.L                        0
.............................
DC.L                 fontName
DC.L                        0
DC.L             fontEnd-font
.............................
DC.L                 fontData
DC.W                        8
DC.L                  fontLoc
DC.L                fontSpace
DC.L                 fontKern

*Hier kommen die Bilddaten
fontData:
DC.W    0x....., 0x......
..........................
*Hier kommen die CharLoc-Daten

fontLoc:
DC.L  .......
```

```
fontSpace:
........................

fontKern:
.........................

* Und hier ist es zu Ende
fontEnd:
END
```

Wenn Sie nun Ihren Zeichensatz so eingetippt haben, dann müssen Sie ihn nur noch assemblieren, linken und unter dem richtigen Namen in der FONTS: Directory abspeichern. Die Datei, die die ***FontContentsHeader*** und die ***FontContents***- Struktur beinhaltet kann übrigens auf die gleiche Weise mit dem Assembler und Linker erzeugt werden.

6. Textausgabe mit IntuiText

Neben der schon im ersten Abschnitt vorgestellten Textausgabeprozedur ***Text***,die der Grafik-Library angehört, bietet auch Intuition eine interessante Prozedur zur Textausgabe. Sie ähnelt stark der schon bekannten ***DrawImage***-Routie, die ein durch eine ***Image***-Struktur definiertes Bild ausgibt. Die Prozedur, von der wir sprechen, heißt ***PrintIText*** und die zugehörige Struktur ***IntuiText***. In dieser Struktur definieren Sie sich einen Text, den Sie dann wiederholt an verschiedenen Bildschirmpositionen ausgeben können. Das Schöne daran ist, daß Sie für diesen Text einen anderen Font und/oder eine andere Schriftart als die eben ge-

nannte verwenden können. Mit Hilfe eines Verkettungszeigers mehrere dieser Strukturen zu einem zusammenhängenden formatierten Text zu verbinden.

Wie dies alles zu machen ist, können Sie der folgenden Beschreibung der *IntuiText*-Struktur entnehmen:

Abb 7.10 Die IntuiText-Struktur

```
struct IntuiText
{
UBYTE FrontPen, BackPen;
UBYTE DrawMode;
SHORT LeftEdge;
SHORT TopEdge;
struct TextAttr *ITextFont;
UBYTE *IText;
struct IntuiText *NextText;
};
```

Die Felder ***FrontPen*** und ***BackPen*** bestimmen, wie den Namen leicht anzusehen ist, die Vorder- und Hintergrundfarbe. Der Zeichenmodus wird durch ***DrawMode*** bestimmt. Dort können Sie die gleichen Werte, die an die ***SetDrMd***-Prozedur übergeben werden, einsetzen. Die nächsten beiden Felder, ***LeftEdge*** und ***TopEdge*** geben den Abstand des Textes von der Bildschirmposition, die beim Aufruf von ***PrintIText*** angegeben wurde, an.

Der Font, der für die Ausgabe des Textes benutzt wird, wird durch die ***TextAttr***-Struktur, auf die ***IText Font*** zeigt, spezifiziert. Wichtig ist, daß der ausgesuchte Zeichensatz in der Systemfontliste verfügbar ist. Falls Sie den Standardfont des Rastports verwenden wollen, dann geben Sie hier NULL an. Der auszugebende Text selbst wendet sich an die Adresse, die in ***IText*** steht. Er muß selbstverständlich durch eine 0 agbeschlossen sein. Wenn Sie mehrere ***IntuiText***-Struk-

turen verketten wollen, dann können Sie dies über *NextText* tun. Schreiben Sie dort einfach die Adresse der nächsten Struktur ein. Um eine solche Kette auszugeben, brauchen Sie, nachdem Sie die einzelnen Strukturen initialisiert haben, nur noch die *PrintI Text* Prozedur wie folgt aufzurufen:

PrintIText(Rast,IText,x,y)

In Text übergeben Sie der Prozedur die Adresse der ersten *IntuiText*-Struktur. Die Koordinaten, die in *x, y* übergeben werden, bestimmen die Position an der Text erscheinen soll. Dabei ist zu beachten, daß er nur dann genau an der Stelle *x,y* erscheint, wenn die *LeftEdge* und *TopEdge* Felder von *IntuiText* beide Null sind. Diese werden nämlich zu *x,y* addiert, um die endgültige Textposition zu ermitteln. Dadurch haben Sie die Möglichkeit, mit mehreren verketetten *Intui Text*-Strukturen, die die entsprechenden *LeftEdge*- und *TopEdge*-Werte haben, einen ganzen Textbildschirm, mit verchiedenen Fonts, Schriftarten und Farben aufzubauen und immer wieder mit einem einzigen Aufruf von *PrintIText* auszugeben.

Zum Abschluß des Kapitels noch ein kurzes Beispiel für die Anwendung von *IntuiText*. In dem nachfolgenden Programm wird ein Text auf die soeben beschriebene Weise dreimal ausgegeben. Dabei wird beim ersten mal der Standardfont des Rastports, beim zweiten der Font "topaz 9" und beim dritten "topaz 8" verwendet.

Da es sich bei beiden um ROM-Fonts handelt, brauchen Sie nicht extra geöffnet zu werden. Normalerweise wird einer der beiden Zeichensätze auch als Standardfont verwendet, so daß Sie sich nicht wundern müssen, sich zu entscheiden. Auch der Unterschied zwischen den beiden "topaz" Fonts hält sich in Grenzen, so daß er vielleicht erst nach genauerem hinse-

hen auffält. Wir haben uns trotzdem entschlossen, sie zu verwenden, um die Anwendung der ROM-Fonts zu zeigen.

Programm 7.5 IntuiText

```
#include "intuition/intuition.h"
#include "graphics/gfx.h"

struct Window   *Window;
struct RastPort *Rast;

struct IntuiText InText = {1,0,
         /*  Vorder und Hintergrundfarbe  */
                         JAM1,
         /*  Zeichenmodus                 */
                         0,0,
         /*  Linker und rechter Offset    */
                         NULL,
         /*  Default Font benutzen        */
                         NULL,
         /*  Noch kein Text               */
                         NULL
         /*  Kein Nachfolger              */
                        };

struct TextAttr ITFont = {"Topaz.Font",
                 /*  Daten des zu öffnenden Fonts  */
                        8,
                 /*  Dies ist ein Standardfont     */
                        0,
                 /*  der sich im ROM-befindet      */
                        FPF_ROMFONT
                       };
```

```
main ()
{
 SHORT  i;
 ULONG  Class, Styles, OldStyle;


 OpenIntui();
        /*  Graphics- und Intuitionlibrary öffnen   */
 OpenGfx();

 /*  Fenster öffnen un die Adresse des Rastports
                                          lesen.    */
 Window = (struct Window*)MakeWindow(200,10,200,
                  50,0,0,"IntuiText",SMART_REFRESH|
                     WINDOWCLOSE,CLOSEWINDOW,NULL);
 if(Window == NULL)
                       /*  Fehler beim Öffnen.      */
   exit(FALSE);
 Rast = Window->RPort;

 InText.IText = "Ganz normaler Font";
                 /*  Auszugebenden Text eintragen   */

 PrintIText(Rast,&InText,10,15);
                 /*  und ausgeben                   */

 InText.IText = "Topaz Größe 9 ";
                 /*  Fontname ändern,               */

 InText.ITextFont = &ITFont;
                 /*  in Inuti Text  eintragen       */

 PrintIText(Rast,&InText,10,25);
                 /*  und Text ausgeben              */

 InText.IText = "Topaz Größe 8";
                 /*  Der nächste Font.              */
```

```
ITFont.ta_YSize = 8;
                 /* Größe  einstellen,                */

InText.ITextFont = &ITFont;
                 /* In IntuiText eintragen            */

PrintIText(Rast,&InText,10,35);
                 /* und Text  ausgeben                */

/* Auf Close Gadget warten und Fenster schließen.*/
Class = WaitEvent(Window,&code);
CloseWindow(Window);

}
```

Kapitel 8

Die Sonderdarstellungsmodi des Amiga

Nachdem Sie in den vorangegangnen Kapiteln die Grundlagen der Grafikprogrammierung am Amiga kennengelernt haben, wollen wir uns hier einigen Besonderheiten zuwenden. Erst diese Sondermodi, mit den wir uns in diesem Kapitel beschäftigen wollen, heben die Grafik des Amiga wesentlich unter den meisten anderen Computern hervor. Linien und Kreise mit einer halbwegs passablen Auflösung konnten schon Computer in der Klasse des C-64 zeichnen. Besonderes der ***HAM***-Modus, der eine gleichzeitige Darstellung aller 4096 Farbe ermöglicht, sollte das Herz eines jeden Grafikprogrammierers höher schlagen lassen. Dazu werden Sie in diesem Kapitel noch erfahren, wie Sie eine extrabreite Anzeige (bis 1024x1024 Pixel), eine Unterteilung des Displays in zwei Tiefenstufen (Dual-Playfield) und einige andere interessante Effekte ausnutzen können. An vielen Stellen dieses Kapitels wird vorausgesetzt, daß der Leser mit dem Inhalt der vorangegangenen Kapitel, vor allem der über Intuition (Kapitel 2) und Farbeinstellung (Kapitel 3), vertraut ist.

1. Der Interlace-Modus

Diesen Modus haben wir schon ganz am Anfang dieses Buches im Zusammenhang mit den verschiednen Auflösungen eines Screens erwähnt. Er kann durch setzen des ***INTERLACE*** Flags in dem **ViewModes**-Feld der **NewScreen**-Struktur beim Eröffnen eines Screens eingeschaltet werden (siehe Kapitel 2). Dadurch wird die vertikale Auflösung des Sreens verdoppelt (kann also bei der deutschen Version des Amiga bis zu 512 Zeilen betragen).

Leider hat die Sache einen Haken: die Verdoppelung der Anzahl der Zeichen wird dadurch erreicht, daß beim Zeichnen des Bildes auf dem Monitor immer abwechselnd die geraden und die ungeraden Zeilen durchlaufen werden. So ergibt sich sowohl für die geraden, als auch für die ungeraden Zeilen, eine Bildwiederholungsfrequenz, die um die Hälfte geringer ist als im normalen Modus. Die logische Folge davon ist, daß das Bild zu flackern beginnt. Besonders schlimm wird es, wenn sie nicht nur Bilder, sondern vorwiegend Text darstellen wollen, denn dieser ist nur mit einiger Anstrengung zu lesen. Ganz vermeiden läßt sich dieser Effekt eigentlich nur durch den Kauf eines entsprechend guten (und teuren !!) Monitors. Aus diesem Grund sollte man, bevor für ein Programm den Interlace-Modus wählt, erst mal überlegen ob dies wirklich unvermeidlich ist. Haben Sie sich einmal für die Interlace-Darstellung entschieden, dann sollten Sie zumindest darauf achten, von einer Zeile zur anderen keine plötzlichen Farbwechsel zu verwenden. Der Flimmereffekt ist nämlich an solchen Stellen besonders deutlich. Statt dessen können Sie bei dem Übergang von einer Farbe zur anderen stufenweise vorgehen. Dazu legen Sie zwischen die beiden Zeilen, einige Zeilen, die Mischfarben beinhalten, so daß ein sanfter Wechsel entsteht. Auf die Weise läßt sich das Flimmern auf ein erträgliches Niveau reduzieren.

2. Mehr Farben - der Extra-Half-Brigth-Modus

Da zu einem Screen bekanntlich höchstens 32-Farbregister gehören können, sind normalerweise auch nicht mehr verschiedene Farben gleichzeitig darstellbar. Um diese Anzahl jedoch zu verdoppeln, haben sich die Amiga-Entwickler einen raffinierten Trick einfallen lassen. Durch Verwendung einer sechsten Bitplane besteht die Möglichkeit, jede dieser 32 Farben mit der halben Helligkeit darzustellen (daher auch der Name, Half-Bright = "halbhell"). Diese Möglichkeit ist vor allem bei der Darstellung dreidimensionaler Objekte oder Schatten besonders nützlich. Die Farben werden im Extra-Half-Brigth-Modus ganz normal wie bei einer Anzeige der Tiefe 5 gewählt. Zu der Nummer der Farbe wird dann, falls diese in der halben Helligkeit erscheinen soll, 32 addiert. Dies entspricht dem Setzen des entprechenden Bits in der sechsten Bitplane. Für die Farbe Nr. 15 ergibt sich dann die Nummer 32 + 15 also 47. Wenn Sie diese Farbe nun zum Zeichnen gebrauchen wollen, können Sie sie wie immer mit ***SetAPen*** bzw. ***SetBPen*** an die Vorder- bzw. Hintergrundfarbe eines Rastports zuweisen.

Um den Extra-Half-Brigth Modus einzuschalten, müssen Sie nur einen Screen der Tiefe 6 aufmachen. Wie auch schon bei einer Tiefe von 5 ist dabei die Auflösung auf lowres, also höchstens 320x256 Pixel beschränkt.

3 Noch mehr Farben - der HAM-Modus

Die 64 Farben, die im Extra-Half-Bright Modus verfügbar sind, sollten eigentlich für die meisten Zwecke genügen. Sie sind aber noch nicht das Beste, was Ihr Amiga zu bieten hat! Mit einigen Einschränkungen ist es nämlich möglich, im HAM-Modus alle 4096 Farben nebeneinander auf den Bildschirm zu bringen. In diesem Modus werden sechs Bitplanes bei einer Auflösung von 320 x 200 Punkten verwendet. Die Einschränkungen bestehen darin, daß sich zwei horizontal benachbarte Punkte nur in einer der drei Komponenten: Rot, Grün oder Blau unterscheiden dürfen. Da man eine solche Farbvielfalt in der Regel sowieso nur dazu braucht, sanfte Farübergänge zu erreichen (z.B. um durch eine ensprechende Schattierung einen 3-D Effekt zu erzeugen), ist diese Einschränkung nicht so bedeutend wie man vielleicht glauben könnte. Um zu verstehen, wie sie zustande kommt, muß man zunächst die Farbdarstellung im HAM-Modus betrachten. Da eine direkte Speicherung der 4096 Farbnummern in Bitplanes nicht möglich ist (man bräuchte dafür 12 Bitplanes und ungefähr 750 KByte), haben sich die Amiga Entwickler wieder mal einen Trick einfallen lassen. Von den sechs Bitplanes, die für HAM-Darstellung gebraucht werden, werden die zwei oberen (Bitplane 5 und 6, also Bits 4 und 5) als "Schalter" benutzt. Sie bestimmen, welche Bedeutung den restlichen Bitplanes bei der Bestimmung der Farbe eines Punktes zukommt. Dabei gibt es die folgenden vier Möglichkeiten.

(1) Die Bits 0 bis 3 (Bitplanes 1 bis 4) werden als Nummer eines der Farbregister 0 bis 15 interpretiert. Der Punkt nimmt dann die durch dieses Register bestimmte Farbe an.

(2) Der rote und der grüne RGB-Anteil der Farbe des Punktes werden von dem linken Nachbarn übernommen. Der Blauanteil wird durch die Bits 0 bis 3 bestimmt (mit 4 Bits kann man gerade die Zahlen 0 bis 15 darstellen).

(3) Der blaue und der grüne RGB-Anteil der Farbe des Punktes werden von dem linken Nachbarn übernommen. Der Rotanteil wird durch die Bits 0 bis 3 bestimmt.

(4) Der blaue und der rote RGB-Anteil der Farbe des Punktes werden von dem linken Nachbarn übernommen. Der Grünanteil wird durch die Bits 0 bis 3 bestimmt.

Diese vier Modi werden durch die folgenden vier Kombination der Bits 4 und 5 ausgewählt:

Abb 8.1 Die Bedeutung der Bits 4 und 5 im HAM-Modus

Bits	Modus
00	1
01	2
10	3
11	4

Wie sieht also die Farbzuweisung im HAM-Modus praktisch aus? Wir wollen uns das mal an Hand einiger konkreter Beispiele ansehen. Als erstes soll einem Punkt einfach eine von 16 vordefinierten Farben zugewiesen werden. Wie der obigen Aufzählung zu entnehmen ist, müssen Sie dazu die oberen beiden Bitplanes (also die oberen beiden Bits der Farbnummer) auf 00 setzen, und mit den unteren vier die Farbe als eine Zahl zwischen 0 und 15 angeben. Haben Sie sich z.B. für die Farbe Nr. 10 entschieden, ergibt sich der Farbwert aus:

$$0 + 10 = 10$$

Dabei wird durch die 0 der Modus 1 und durch die 10 das entsprechende Farbregister angewählt. Diese Farbe kann dann wie schon bekannt mit **SetAPen** zur Vordergrundfarbe gemacht werden. Mit den in den früheren Kapiteln beschriebenen Zeichenroutinen können Sie dann beliebigen Punkten Ihrer Anzeige diese Farbe zuordnen.

Einfach, nicht wahr? Aber natürlich auch langweilig, denn das können Sie auch mit einem normalen Screen der Tiefe 4 machen!

Als nächstes wollen wir mal in einer Zeile von links nach rechts einen kleinen Regenbogen zeichnen. Dabei soll der erste Punkt von Links schwarz sein (RGB-Zusammensetzung 0,0,0) und der letzte die Farbe der RGB-Zusammensetzung 15,15,15 haben. Dabei soll von links nach rechts zunächst der Blauanteil, dann der Rotanteil und schließlich der Grünanteil bis 15 ansteigen. Insgesamt wären das 15 + 15 + 15 also 45 verschiedene Farben in einer Zeile. Nachdem Sie den ersten Punkt auf die gerade beschriebene Art schwarzgemacht haben, wollen Sie für den daneben liegenden Punkt die grüne und die rote Farbkomponente

übernehmen (also 0 lassen !) und die blaue auf 1 setzen. Die unteren 4 Bits der Farbnummer müssen also den Wert 0001 haben (Binärzahl 1), während die oberen beiden 01 sein müssen. Das ergibt das folgende Bitmuster:

01 0001

Die Farbnummer ist also:

1 * 2 Hoch 4 + 1 = 16 + 1 = 17

Bei den nun nachfolgenden Punkten setzen sie analog den Blauanteil immer höher (die Farbnummern sind dann 18, 19 etc. Es verändern sich also nur die 4 unteren Bits) bis dieser bei 15 angelangt ist. Nun ist es an der Zeit den Rotanteil zu verändern. Die oberen Bits müssen hierzu 10 sein, die unteren durchlaufen wieder alle die Zahlen 1 bis 15 (also binär 0001 bis 1111). Die Farbnummer für den sechszehnten Punkt von links ist demnach:

1 * 2 Hoch 5 + 1 = 32 + 1 = 33

Bei den Punkten 30 bis 44 wird analog der Rotanteil schrittweise vergrößert wobei zu beachten ist, daß die beiden oberen Bits nun 11 sind. Auf eine ähnliche Art und Weise werden in dem folgenden Beispielprogramm 256 Farben als Linien nebeneinander dargestellt.

Programm 8.1 HAM

```
#include"Display.h"
#include"graphics/gfx.h"
#include"intuition/intuition.h"
struct Window     *Window;
struct Screen     *Screen;
struct RastPort   *Rast;
struct ViewPort   *View;


main ()
{
 USHORT  code, i, j, h; ULONG   Class;

 OpenIntui();
 OpenGfx();

 /*  Einen low-res Screen öffnen.  */
 Screen = (struct Screen *)MakeScr(0,0,320,250,
                              "HAM-Demo",6,HAM,NULL,NULL);
 if (Screen == NULL)
  exit(FALSE);

 /*  Ein Fenster auf dem neuen Screen öffnen   */
 Window = (struct Window *)MakeWindow(20,50,270,
           100,0,0,"Ham-Demo",WINDOWCLOSE | ACTIVATE,
           MOUSEBUTTONS | CLOSEWINDOW,Screen);
 if(Window == NULL)      /*  Fehler beim Öffnen ?  */
  exit(FALSE);

 /* Adresse des Viewports und des Rastports holen  */
 Rast = Window->RPort;
 View = &((*(Window->WScreen)).ViewPort);

 SetRGB4(View,0,5,0,0);
                         /*  Hintergrundfarbe schwarz  */
  for(i = 0; i <= 15; i++)
```

```
{
 for(j = 0; j< 15; j++)
 {
  SetAPen(Rast,32+j);
  Move(Rast,i*16+j,0);
  Draw(Rast,i*16+j,100);
 };
 SetAPen(Rast,16+i);
 Move(Rast,i*16+15,0);
 Draw(Rast,i*16+15,100);
};


  /* Auf Close-Gadget warten und alles schließen */

 Class = WaitEvent(Window,&code);
 CloseWindow(Window);
 CloseScreen(Screen);
}
```

Da die Farbe eines Punktes im HAM-Modus beim Zeichnen nicht nur von der aktuellen Vorder- bzw. Hintergrundfarbe, sondern auch von der Farbe seiner Nachbarn abhängt, sollte man mit Prozeduren, die ganze Figuren zeichnen (also z.B. **Draw** oder **DrawCircle**) vorsichtig umgehen. Es ist nämlich nicht unbedingt gesagt, daß die gesamte Figur dann in der gewünschten Farbe erscheint. Ein weiteres Problem ergibt sich aus der Tatsache, daß der erste Punkt von links (also der mit der x-Koordinate 0) keinen linken Nachbarn hat. Dies heißt, daß man seine Farbe immer direkt, also im Modus 1, setzen muß.

4. Erstellen eigener Viewports

Bevor wir mit der Beschreibung weiterer Darstellungsmodi fortfahren, wollen wir den Bilddarstellungsmechanismus noch einmal genauer unter die Lupe nehmen. Wir wollen dabei versuchen, ein Bild ganz ohne Zuhilfenahme von Intuition, also ohne Screens und Fenster, zu erzeugen. Dazu müssen wir zuerst die ***ViewPort-*** und ***View-***Strukturen der Grafik-library und ihre genaue Funktion untersuchen. Das gesamte Bild, das zu einem gegebenen Zeitpunkt angezeigt wird, wird durch die **View**-Struktur bestimmt. Diese umfaßt dann mehrere **ViewPort**-Strukturen, die ihrerseits die einzelnen Displayelemente beschreiben. Die **View** an sich besagt weder etwas über die Auflösung, noch über die Größe der Anzeige. Sie beinhaltet lediglich Informationen über die Anordnung der Viewports. So kann das Bild zu jedem Zeitpunkt aus mehreren Ausschnitten (Viewports) verschiedener Auflösung und Größe bestehen. Genau das sehen Sie, wenn Sie mehrere Screens verschiedener Auflösung übereinander schieben. Jedem Screen ist nämlich eine **ViewPort**-Struktur, die seine Eigenschaften bestimmt, zugeordnet. Diese wird dann, wenn der Screen sichtbar wird, von Intuition an die Liste der **ViewPort** der aktuallen **View** angehängt. Soll der Screen wieder verschwinden, dann wird dieser Eintrag einfach wieder entfernt. Neben View und ViewPorts sind für den Aufbau eines Bildes noch zwei weitere, aus früheren Kapiteln schon bekannte, Strukturen notwendig: die **BitMap-** und die **RastPort**-Struktur. Die erste stellt die Verbindung zu dem Speicherbereich her, in dem die Bilddaten liegen, während die zweite bekanntlich den "Zeichenstift" beschreibt. Beide gehören zu einem einzelnen **ViewPort**, nicht zu einer **View**. Die Verbindung zwischen dem **ViewPort** und

der **BitMap** stellt eine weitere Struktur, nämlich **RasInfo** her. So beinhaltet **ViewPort** einen Zeiger auf **RasInfo** und **RasInfo** einen Zeiger auf die **BitMap.** Die Adresse der **BitMap** ist selbstverständlich auch im **RastPort** zu finden. Einen Überblick über die beschriebenen Beziehungen dieser Strukturen können Sie sich auch anhand der nachfolgenden Abbildung verschaffen:

Bild 8.1 - Die Verknüpfung der View-Strukturen.

Aus der obigen Beschreibung ergibt sich nun die folgende Vorgehensweise beim Erstellen einer eigenen Anzeige:

(1) Erzeuge und initialisiere eine **View**-Struktur.

(2) Für jedes Element der Anzeige erzeuge und initialisiere eine **ViewPort**-Struktur.

(3) Für jede **ViewPort**-Struktur erzeuge eine **RasInfo**-Struktur und trage ihre Adresse in **ViewPort** ein.

(4) Für jedes Anzeigeelement muß eine **BitMap** (die eigentliche **BitMap**-Struktur und der dazu gehörige Speicher für Bilddaten) erzeugt und in die entsprechende **RasInfo**-Struktur eingetragen werden.

(5) Damit in jedes Element der Anzeige auch mit Hilfe der Grafikroutinen der Grafik-Library gezeichnet werden kann, müssen noch **RastPort**-Strukturen erzeugt und initialisiert werden. In jeden **RastPort** muß dann die Adresse der dazu gehörigen **BitMap** eingetragen werden. Falls Sie kein Rastport erzeugen, müssen Sie durch direktes Schreiben in den Bilddatenspeicher zeichnen, was recht mühsam ist.

(6) Bringe die eigenen View zur Anzeige.

Dies ist selbstverständlich nur ein grober "Schlachtplan" der noch näherer Erläuterung bedarf. Wir werden allerdings dabei auf die genaue Beschreibung der einzelnen recht umfangreichen Strukturen im Rahmen dieses Kapitels verzichten. Deren genauen Beschreibung finden Sie im Anhang B. Der erste Punkt, die Initialisierung der **View**-Struktur, ist recht einfach. Sie brauchen nur eine Variable des Typs **struct View** zu vereinbaren:

struct View View;

und übergeben Ihre Adresse wie folgt an die Prozedur **InitView**:

InitView(&View);

Diese trägt in alle Felder der Struktur die Werte 0 bzw. -1 ein. Als nächstes werden analog, diesmal mit Hilfe der **InitVPort**-Prozedur, die **ViewPorts** erzeugt. Sie müssen dazu noch einige Felder der **ViewPort**-Strukturen wie folgt initialisieren:

***DWitdh,DHeight*:**
Gewünschte Höhe und Breite des neuen Viewports.

***DxOffset, DyOffset*:**
Die Lage des neuen Viewports im Bezug auf die linke obere Ecke der Anzeige. Zwischen zwei Viewports muß immer mindestens eine Bildzeile Abstand gelassen werden. Das heißt auch, daß Viewports zwar untereinander, nie aber nebeneinander liegen können.

***Modes*:**
Der Anzeigemodus, also die Auflösung und die gewünschten Sondermodi. Hier können Sie die gleichen Werte wie im Kapitel 2 für das **ViewModes**-Feld der **NewScreen**-Struktur beschrieben, eingesetzt werden (Also z.B. HIRES ¦ MAM).

***ColorMap*:**
Hier wird die Adresse einer **ColorMap**-Struktur, die die RGB-Zusammensetzung der für den Viewport gültigen Farben beinhaltet, eingetragen.

***RasInfo*:**
Die Adresse der dazu gehörigen **RasInfo**-Struktur. Wie diese initialisert werden muß, wird später beschrieben.

Eine korrekt initialisierte **ColorMap**-Struktur bekommen Sie, wenn Sie die Routine ***GetColorMap***(n) aufrufen. Sie liefert als Funktionsergebnis die Adresse einer solchen, vollständig initialiserten Struktur, die n Farben aufnehmen kann. An die

Adresse,die durch das **ColorTable**-Feld dieser Struktur angegeben wird, können Sie dann die gewünschten Farbwerte kopieren. Die kann im Programm etwa wie folgt aussehen:

```
struct ColorMap *CMap;
.........
short Colors [] = {0x000, 0xf00, 0x0f0, 0x00f};
.........
CMap = GetColorMap(4);
CopyMem(Colors,CMapB>ColorTable,8);
```

Beachten Sie dabei, daß die Anzahl der zulässigen Farben von der Tiefe der Bitmap Ihres **Viewports** abhängt (siehe unten). Um den Viewport ganz fertigzustellen, fehlt nur noch die **RasInfo**-Struktur. Bevor Sie in den Viewport eingetragen wird, müssen die folgenden Felder initialisiert werden:

Next:
Dieses Feld hat nur für den *DualPlayfield*-Modus eine Bedeutung und sollte sonst unbedingt auf NULL gesetzt werden.

RxOffset, RyOffset:
Die Position der linken oberen Ecke der Bitmap in Bezug auf die linke obere Ecke des Viewports. Diese Felder können, wie Sie später sehen werden, sehr gut zum Scrollen benutzt werden.

BitMap:
Hier wird die Adresse der **BitMap**-Struktur des Viewports eingetragen. Die **BitMap**-Struktur, deren Adresse in das letzte Feld von **RasInfo** eingetragen werden muß, beinhaltet Angaben zur Tiefe, Breite und Höhe der Bitmap, sowie eine Tabelle mit den Adressen der

Bitplanes (also der Speicherbereiche, wo die eigentliche Bildinformation gespeichert ist). Um sie zu erstellen müssen Sie zweierlei tun: Zuerst rufen Sie die Prozedur **InitBitMap** mit der Adresse einer nicht initialisierten **BitMap**-Struktur (**BMap**), der Tiefe (**Depth**), der Breite in Pixel (**Width**) und der Höhe (**Height**) wie unten dargestellt, auf.

```
InitBitMap(BMap,Depth,Width,Height);
```

Wichtig ist, daß dieser Aufruf lediglich die Felder der Struktur initialisiert, und keine neue **BitMap**-Struktur erzeugt. Auch die Allozierung des für die Bitplanes benötigten Speicherbereiche und das Eintragen ihrer Adressen in das **Planes[]**-Feld der Struktur müssen Sie selbst in die Hand nehmen. Dies geschieht am besten in einer **for**-Schleife mit Hilfe der schon bekannten AllocMem-Routine.

```
for(i = 1; i <= Depth; i++)
BMap->Planes[i-1]=AllocMem((Width*Height),
MEMF_CHIP|MEMF_CLEAR));
```

Vergessen Sie dabei nicht, daß diese Speicherbereiche im CHIP-RAM liegen müssen. Es ist auch durchaus sinnvoll, durch Setzen des **MEMF_CLEAR**-Flags den Speicher gleich mit Nullen zu füllen (sonst werden Sie nach dem Einschalten des Viewports irgendeinen "Müll" zu sehen bekommen). Nachdem alle zu der **View** gehörenden Strukturen wie beschrieben erstellt wurden, kann sie endlich auf den Bildschirm gebracht werden. Dazu sind nur noch die folgenden drei Aufrufe notwendig:

```
MakeVPort(&View,&ViewPort);
MrgCop(&View);
LoadView(&View);
```

Die ersten beiden sorgen dafür, daß die Befehlslisten für den Coprozessor (Copperlisten), die für die Bilddarstellung notwendig sind, erstellt werden (siehe Kapitel 10). Eingeschaltet wird unsere View mit dem letzten Befehl. Er bewirkt, daß alles was bisher auf dem Bildschirm zu sehen war, verschwindet und durch unsere, zunächst leere Anzeige ersetzt wird. Zum Zeichnen in den Viewports können Sie entweder direkt in die Bitplanes schreiben, oder zu jedem Viewport einen eigenen Rastport erzeugen, indem Sie alle Grafikbefehle anwenden können. Zur Erstellung des Rastports übergeben Sie einfach die Adresse einer **RastPort**-Struktur an die **InitRastPort**-Prozedur, und tragen dann in das **BitMap**-Feld der Struktur die Adresse der zu dem Viewport gehörenden **BitMap**.

Das sieht dann im Programm z.B. so aus:

```
InitRastPort(&Rast);
Rast.BitMap = ViewPort.RasInfo.BitMap;
```

An dieser Stelle noch ein Tip. Falls Sie irgendwann wieder zu der alten Anzeige (also z.B. zum Workbench-Screen) zurückkehren wollen, dann sollten Sie sich vor dem Aufruf von **LoadView** die Adresse der aktuellen **View** merken. Sie können sie wie folgt aus der **GfxBase**-Struktur auslesen:

```
oldView = GfxBase->ActiView;
```

Neben der Wiederherstellung des alten Bildschirminhaltes sollten Sie auch an die Freigabe des nicht mehr benötigten Speichers denken. Dabei handelt es sich um die Bitplanes, die **ColorMaps** und die Copperlisten. Die Deallozierung der Bitplanes geschieht in einer zu der Allozierung analogen Weise mit Hilfe der **FreeMem**-Prozedur. Die **ColorMaps** und die Copperlisten der einzelnen Viewports können folgendermaßen mit Hilfe der **FreeColorMap** und **FreeVPortCopLists** Routinen freigegeben werden.

```
FreeVPortCopLists(&ViewPort);
FreeColorMap(&ColMap);
```

Als letztes muß nur noch die Copperliste der gesamten **View** mittels **FreeCprList** so gelöscht werden:

```
FreeCprList(View.LOFCprList);
```

Zum Abschluß dieses recht langen Abschnittes ein Beispielprogramm. Es erzeugt eine View **(V)** mit zwei untereinanderliegenden Viewports (**View1** und **View2**) verschiedner Auflösung (320 x 200 und 640 x 200). Interessant ist dabei die Tatsache, daß beide Viewports die gleiche **RasInfo**-Struktur, also auch die gleiche Bitmap haben. Dadurch wird alles, was in den einen Viewport gezeichnet wird, auch im anderen sichtbar ist, allerdings in einer anderen Auflösung. Um diesen Effekt zu demonstrieren, wird durch die Programmeigene Prozedur **CreateDisplay** in den ebenfalls gemeinsamen Rastport der Vieports eine aus vielen Ellipsen bestehende Figur langsam gezeichnet. Sie können dies deutlich beobachten, da die Figur in den beiden Viewports gleichzeitig entsteht.

Zur Erzeugung und Freigabe der Bitmap benutzt das Programm Prozeduren, die wir in einem späteren Abschnitt im Zusammenhang mit einem neuem include-File vorstellen werden.

Programm 8.2 Viewports

```
#include  "exec/types.h"
#include  "exec/memory.h"
#include  "graphics/gfx.h"
#include  "graphics/view.h"
#include  "graphics/copper.h"
#include  "DisplayTools.h"
#define DEPTH   1
#define WIDTH   300
#define HEIGHT  120
#define XO      160
#define YO      130

struct View       V, *oldView;
struct ViewPort   View1, View2;
struct ColorMap   *CMap;
struct RasInfo    RInfo;
struct RastPort   Rast;
struct BitMap     *BMap;
short  Colors [] = {0x00A, 0x0A0};

/*  Farbtabelle: Blau und Grün    */
main()
{

  OpenGfx();

BMap = MakeBitMap(DEPTH,WIDTH,HEIGHT);
                                     /*  BitMap erzeugen */
```

```
/* Initialisiere die View und Viewport
                    Strukturen beider Viewports  */
InitView(&V);
InitVPort(&View1);
V.ViewPort = &View1;
               /* ViewPort und View verknüpfen  */
InitVPort(&View2);
View1.Next  =  &View2;
                /* Nächsten Viewport anknüpfen  */

/* RasInfo initialisieren  */
RInfo.BitMap   = BMap;
RInfo.RxOffset = 0;
RInfo.RyOffset = 0;
RInfo.Next     = NULL;

/* Farbtabelle erstellen  */
CMap = GetColorMap(2);

/* Allozieren  */
CopyMem(Colors,CMap->ColorTable,sizeof(Colors));
/* Farben eintragen  */

/* Alles in den ersten ViewPort und */
View1.RasInfo  = &RInfo;
View1.ColorMap = CMap;
View1.DWidth   = WIDTH;
View1.DHeight  = HEIGHT;

/* in den zweiten ViewPort eintragen  */
View2.RasInfo  = &RInfo;
View2.ColorMap = CMap;
View2.DWidth   = WIDTH;
View2.DHeight  = HEIGHT;
View2.DxOffset = X0;
```

```
  /*  ViewPort nach unten versetzt  */
  View2.DyOffset = YO;
  View2.Modes = HIRES;

  /*  Copperliste für die Viewports erzeugen  */
  MakeVPort(&V,&View1);
  MakeVPort(&V,&View2);
  MrgCop(&V);

  /*  Einen Rastport erzeugen und initialisieren  */
  InitRastPort(&Rast);
  Rast.BitMap = BMap;

  /*  Aktuelle View retten und die eigene in
                                Vordergrund bringen,  */
  oldView = GfxBase->ActiView;
  LoadView(&V);

  /*  etwas in den neuen Viewport zeichnen  */
  CreateDisplay(&Rast);

  /*  und den alten Zustand wiederherstellen  */
  LoadView(oldView);
  FreeAll();

}


FreeAll()
{

  /*   Copperlisten freigeben  */
  FreeVPortCopLists(&View1);
  FreeVPortCopLists(&View2);
  FreeCprList(V.LOFCprList);

  /*  BitMap samt Bitplanes freigeben  */
  FreeBitMap(BMap);
```

```
  /*  Farbtabelle freigeben  */
  FreeColorMap(CMap);

}

CreateDisplay(Rast)

struct RastPort *Rast;

{
  int i,j ;

  /*  Text ausgeben  */
  Move(Rast,70,8);
  SetAPen(Rast,1);
  Text(Rast,"Hallo hier bin ich !!",20);

  /*  Einige Ellipsen zeichnen  */
  for(i = 0; i<100; i++)
  {
    DrawEllipse(Rast,150,70,i,50-i/2);
    for(j = 0; j<6000; j++);
  }
}
```

5. Manipulieren eines Viewports: ScrollVPort

Wenn man sich die **ViewPort**-und die **RasInfo** Strukturen näher anschaut, sieht man, daß es durchaus interessant sein könnte, einige ihrer Felder zu modifizieren. Dabei sind vor allem **DxOffset**, **DyOffset** (in **ViewPort**) bzw. **RxOffset**, **RyOffset** (in **RastPort**) gemeint. Durch Veränderung dieser Felder kann die Position des Viewports innerhalb der View bzw. der Bitmap innerhalb des Viewports verändert werden. Dadurch lassen sich sehr leicht fließende Scrolleffekte erzielen. Dieses Verfahren wird auch von Intuition beim Verschieben und Übereinanderschieben von Screens angewandt. Der Vorteil gegenüber dem bekannten Scrolling mittels **ScrollRaster** liegt darin, daß hierbei Teile des Bildes zwar vom Bildschirm verschwinden können, dabei aber nicht gelöscht werden. Sie können also durch Scrollen um den gleichen Betrag in umgekehrte Richtung den alten Bildschirminhalt wieder herstellen. Das Verändern der Komponenten des Viewports allein reicht allerdings nicht ganz aus, um eine Veränderung des Bildes zu erzielen. Wenn Sie es versuchen, werden Sie feststellen, daß sich nichts tut. Das liegt daran,daß sich die Hardware (der Copper) die für die Bilderstellung relevante Information nicht direkt aus den Datenstrukturen, sondern aus den daraus aufgebauten Copperlisten holt. Um diese zu modifizieren, müssen Sie nach jeder Veränderung in der **ViewPort-** bzw. **RasInfo**-Struktur **ScrollVPort** mit der Adresse der **ViewPort**-Struktur aufrufen. Ein Beispiel für Scrollen nach dem oben beschriebenen Prinzip werden Sie in dem nächsten Programm finden.

6. Der Dual-Playfield Modus

In diesem Modus besteht ein Viewport aus zwei Schichten (***Playfields***). Eine von ihnen befindet sich dabei im Vordergrund, die andere im Hintergrund. Das Besondere ist, daß ein beliebiger Pixel jeder Schicht durchsichtig gemacht werden kann. Es kann auch jedes Playfield unabhängig vom anderen beschrieben werden. Das heißt, daß Sie etwas in die vordere Schicht schreiben können, ohne den Inhalt der hinteren Schicht an der gleichen Position zu löschen. Wenn Sie das erste Playfield später an dieser Stelle wieder durchsichtig machen, kommt der unveränderte Inhalt des Hinteren wieder zum Vorschein. Dies ist besonderes für Spieleprogrammierung sehr nützlich. Sie können z.B. die vordere Schicht bis auf einen kleinen durchsichtigen Kreis schwarz machen und das eigentliche Bild in die hintere Schicht zeichnen. Wenn Sie nun eines der beiden Playfields Scrollen, entsteht der Eindruck eines bewegten Fernrohres.

Wir wollen zuerst die Einrichtung eines Dual-Playfield-Displays auf der Viewport-Ebene betrachten. Später werden Sie anhand eines Beispielprogramms sehen, wie sich ein Dual-Playfield Screen öffnen läßt. Ein Dual-Playfield Viewport unterscheidet sich von einem "normalen" in zwei Punkten:

(1) Das **DUALPF**-Bit des Modes Feldes der **ViewPort**-Struktur ist gesetzt. Falls Sie wollen, daß nicht das erste, sondern das zweite Playfield im Vordergrundliegt, müssen Sie auch das Bit setzen.

(2) Die **ViewPort**-Struktur beinhaltet zwei **RasInfo**-Strukturen, von den jede die Adresse einer anderen

Bitmap beinhaltet. Die Adresse der ersten **RasInfo**-Struktur steht wie immer in dem **RasInfo**-Feld der **ViewPort**-Struktur, die Adresse der zweiten in dem **Next**-Feld der ersten. Jede der **RasInfo**-Strukturen ist für eines der beiden Playfields stellvertretend (daher auch die verschiedenen Bitmaps, für jedes Playfield eine eigene).

Da man gewöhnlich in jede Schicht etwas zeichnen oder schreiben möchte, sollte man auch zwei **RastPort**-Strukturen erzeugen (siehe Abschnitt 5). Jede dieser Strukturen beinhaltet einen Zeiger auf die Bitmap des entsprechenden Playfields. Da sich der Zeiger auf die **ColorMap**-Struktur in der **ViewPort**-Struktur befindet, ist sie für beide Playfields gemeinsam da. Damit trotzdem jedes Playfield seine eigenen Farben benutzen kann, ist die erste Hälfte der Farbtabelle für das erste, und die Zweite für das zweite Playfield reserviert. Wenn also das erste Playfield die Tiefe 1 (also 2 Farben) hat, dann wird die Farbe 1 des zweiten Playfields in Wirklichkeit die Farbe 3 sein. Denken Sie auch daran, daß die Farbe 0 bei bei den Playfields Duchrsichtigkeit bedeutet. Die maximale Anzahl der Farben eines Playfields ist 8, da die Summe der Tiefen der Playfields, also die Tiefe der gesamten Anzeige, höchstens 6 sein darf. Je nach Gesamttiefe der Anzeige können die einzelnen Playfields nur folgende Tiefen haben:

Abb. 8. Die Bitplaneverteilung bei Dual-Playfield

Gesamt Tiefe	Playfield 1 Tiefe	Playfield 2 Tiefe
1	1	0
2	1	1
3	2	1
4	2	2
5	3	2
6	3	3

Zum Schluß noch einmal ein Überblick darüber, was zur Erstellung eines Dual-Playfield-Viewports getan werden muß:

(1) Eine **ViewPort**-Struktur mit gesetzen **DUALPF**-Bit erzeugen.

(2) Zwei Bitmaps (**BitMap**-Strukturen + allozierte Bitplanes) für die beiden Playfields erzeugen. Die Bitmaps müssen nicht unbedingt gleich groß sein.

(3) Zwei **RasInfo**-Strukturen erstellen, über das **Next**-Feld verketten und die Adressen der Bitmaps eintragen.

(4) Die Adresse der ersten **RasInfo**-Struktur in den Viewport eintragen.

(5) Zum Zeichnen für jedes Playfield einen Rastport erzeugen.

Bei dieser Aufzählung haben wir natürlich nur das erwähnt, was speziell für den Dual-Playfield Modus wichtig ist (siehe Abschnitt 5).

7. Öffnen eines Dual-Playfield Screens

Im Allgemeinen ist es bequemer, mit Screens und Fenstern, als mit Viewports zu arbeiten. Dies gilt auch für den Dual-Playfield-Modus. Wenn Sie sich an Kapitel 2 erinnern, dann wissen Sie noch, daß in dem **ViewModes**-Feld der ***NewScreen***-Struktur, die die Displayeigenschaften bestimmt, auch das **DUALPF**-Flag gesetzt werden kann. Man kommt da in Versuchung, dieses Flag erst einmal zu setzen, und zu hoffen, daß nach dem Aufruf von **OpenScreen** ein Dual-Playfield-Screen geöffnet wird. Leider ist die Sache aber nicht ganz so einfach. Es wird zwar das entsprechende Flag im Viewport des Screens gesetzt und eine zweite ***RasInfo***-Struktur erzeugt und angehängt, aber keine zweite Bitmap erstellt. Die ***OpenScreen***-Routine ignoriert auch die Tatsache, daß im Dual-Playfield-Modus die Gesamttiefe der Anzeige unter die Playfields verteilt wird, und trägt eine Bimap der Gesamttiefe in die erste ***RasInfo***-Struktur ein. Um einen Dual-Playfield-Screen zu erzeugen, müssen Sie also die Erstellung und das Eintragen der Bitmaps selbst in die Hand nehmen. Dazu setzen Sie beim Öffnen des Screens zusätzlich das **CUSTOMBITMAP**-Flag des ***Type***-Feldes der ***NewScreen***-Struktur und tragen die Adresse der Bitmap des ersten Playfields in das ***Bitmap***-Feld ein. Dazu müssen Sie die Bitmap selbstverständlich vorher erstellt haben!

Damit eine genügend große Farbtabelle erzeugt wird, geben Sie als Tiefe die Gesamttiefe, also die Summe der Tiefen der Playfields an. Ist der Screen geöffnet, so muß nur noch die Adresse der Bitmap des zweiten Playfields (sie muß vorher natürlich auch erstellt werden !) in die zweite *RasInfo*-Struktur des Viewports des Screens eingetragen werden. Damit das System die zweite Bitmap wahrnimmt, müssen Sie zum Schluß noch die *ScrollVPort*-Prozedur aufrufen. Wenn Screen ein Zeiger auf den neuen Screen ist, dann sieht es so aus:

```
Screen->ViewPort.RasInfo->Next->BitMap = BitMap2;
ScrollVPort(&(Screen->ViewPort));
```

Die **RastPort**-Struktur, die in der **Screen**-Struktur enthalten ist, gehört selbstverständlich zu dem ersten Playfield. Daher sollten Sie sich einen zweiten Rastport für das zweite Playfield erzeugen. Da das Öffnen eines Dual-Playfield-Screens recht mühsam ist, haben wir in dem am Ende des Kapitels abgedruckten include-File eine Routine geschrieben, die dies für Sie tut. Sie benutzt zum Öffnen des Screens die *MakeScreen*-Routine aus dem include-File Display.h aus dem zweiten Kapitel.

Im nachfolgenden Beispielprogramm benutzen wir diese Routine, um einen Dual-Playfield Screen zu Öffnen. Dann wird in jedem Playfield ein Gitter aus kleinen Quadraten erzeugt. Diese Gitter werden dann hin und her gescrollt. Dabei können Sie sehr gut beobachten, daß das eine Gitter sich immer vor dem anderen befindet. Beim Scrollen benutzen wir die im vorigen Abschnitt besprochene Methode des Veränderns der X und Y-Offsets eines Viewports.

Programm 8.3 Dual-Playfield

```
#include"graphics/gfx.h"
#include"intuition/intuition.h"
#include"stdio.h"
#include"DisplayTools.h"

struct Screen     *Screen;
struct RastPort   *Rast1, *Rast2;

main ()
{

 short n,j;
 long i;

 OpenIntui(); OpenGfx();
 /*  Einen low-res Screen öffnen.  */
 Screen = (struct Screen *)MakeDPFScr(0,0,320,250,
          "DualPlayfield",4,NULL,NULL,&Rast1,&Rast2);

 /*  Ein Durchsichtiges Viereck im Playfield 1 an
                                   der Mausposition   */
 SetAPen(Rast1,1);
 SetAPen(Rast2,1);
 for (i = 5; i <= 15; i++)
  for (j = 3; j <= 12; j++)
  {
    RectFill(Rast1,15*i,15*j,15*i+8,15*j+8);
    RectFill(Rast2,15*i+5,15*j+5,15*i+12,15*j+12);
  };


 for (i = 0; i <= 5; i++)
 {
  for (j = 0; j <= 50; j++)
  {
```

```
    (Screen->ViewPort).RasInfo->Next->RxOffset += 1;
    ScrollVPort(&(Screen->ViewPort));
   };
   for (j = 0; j <= 100; j++)
   {
    (Screen->ViewPort).RasInfo->Next->RxOffset -= 1;
    ScrollVPort(&(Screen->ViewPort));
   };
   for (j = 0; j <= 50; j++)
   {
    (Screen->ViewPort).RasInfo->Next->RxOffset += 1;
    ScrollVPort(&(Screen->ViewPort));
   };
  };

  /* Screen schließen */
  CloseDPFScreen(Screen);
}
```

Das Beispielprogramm und die obige Beschreibung bezogen sich auf einen Screen. Was ist aber wenn man im Dual-Playfield Modus die Vorteile der Fenstertechnik nutzen will (Menus, IDCMP, etc.) ? Nun, da ein Fenster weder einen eigenen Viewport noch (in der Regel) eine eigene Bitmap besitzt, kann man verständlicherweise kein Dual-Playfield-Fenster auf einem normalen Screen öffnen. Andererseits spricht aber nichts dagegen, ein Fenster auf einem Dual-Playfield Screen zu öffnen. Beachten muß man dabei nur, daß Rahmenelemente bzw. Gadgets nicht die Farbe 0 haben, also nicht durchsichtig sind.

8. "Übergroße"-Anzeige

Die Größe des Bildes, das auf dem Bildschirm zu sehen ist, ist wie Sie bereits wissen, auf höchstens 620x400 Pixel begrenzt. Die Betonung liegt dabei allerdings auf den Zusatz "das zu sehen ist". Wenn Sie mit eigenen Viewports oder mit Screens mit einer eigenen Bitmap arbeiten, dann kann Ihr Bild unabhängig von der Auflösung bis zu 1024x1024 Pixel groß sein. Welcher Teil des Bildes gerade gezeigt wird, können Sie durch die **RxOffset** und **RyOffset**-Felder der zugehörigen **RasInfo**-Struktur (siehe Programm 8.3) bestimmen. Da diese beiden Felder die Lage der Bitmap innerhalb des Viewports angeben, können Sie durch Angabe entsprecheder Werte einen beliebigen Ausschnitt zur Anzeige bringen. Diese Technik ist besonderes für sehr schnelles fließendes Scrolling, wie es etwa bei Action-Spielen benötigt wird, sehr gut geeignet. Das entscheidene dabei ist, daß nicht immer wieder nach jedem Scrollschritt das Bild am Rande aufgebaut werden muß.

9. Das DisplayTools.h include-File

Dieses include-File beinhaltet Prozeduren, die Ihnen das Arbeiten mit den verschiedenen Sondermodi und miteigenen Viewports erleichtern. Die erste, ***MakeBitMap*** dient dazu, eine Bitmap der Tiefe **Depth**, der Höhe **Height** Zeilen und der Breite **Width** Pixel zu erstellen und gibt einen Zeiger auf die ***BitMap***-Struktur zurück. Dabei wird auch der für die

Bitplanes notwendige Speicher alloziert und in das Planes-Feld der Struktur eingetragen. Um eine so erstellte Bitmap samt Bitplanes wieder freizugeben, können Sie die nächste Routine: ***FreeBitMap*** benutzen. Ihr wird lediglich der Zeiger auf die ***BitMap***-Struktur übergeben.

Die Routine ***MakeDPFScr*** hilft Ihnen, einen Dual-Playfield-Screen wie in Abschnitt 7 beschrieben zu Öffnen. ***MakeDPFScreen*** gibt als Funktionsergebnis die Adresse des fertigen Dual-Playfield-Screens zurück und schreibt in **Rast1** und **Rast2** die Zeiger auf die Rasports der beiden Playfields. Als Tiefe müssen Sie die Gesamttiefe, übergeben. Zum schließen eines so erzeugten Screens haben wir die ***CloseDPFScreen*** Prozedur bereitgestellt. Zum Abschluß noch eine Übersicht über die 4 Routinen und ihre Parameter:

Eine Bitmap erzeugen.

MakeBitMap(Depth, Width, Height)
int Depth, Width, Height;

Eine Bitmap freigeben.

FreeBitMap(BMap)
struct BitMap *BMap;

Einen Dual Playfield Screen öffnen.

```
MakeDPFScr (x, y, w, h, Name, d, flags,
font, Rast1,Rast2)
struct RastPort *(*Rast1), *(*Rast2);
short           x, y, w, h, d;
LONG            flags, font;
char            *Name;
```

.Einen Dual-Playfield Screen schließen.

```
CloseDPFScreen(Screen)
struct Screen *Screen;
```

Programm 8.4 Dual-Playfield

```
#include "graphics/gfx.h"
#include "graphics/gfxbase.h"
#include "exec/memory.h"
#include "Display.h"#include "stdio.h"

MakeBitMap(Depth, Width, Height)

int Depth, Width, Height;

{
  struct BitMap *BMap;
  int           i;

 BMap = AllocMem(sizeof(struct BitMap),NULL);
 if (BMap == NULL)
  exit(FALSE);                 /* Fehler  */
  /* BitMap-Struktur initialisieren  */
  InitBitMap(BMap,Depth,Width,Height);

  Width = (Width +7) /8;

  /* Speicher für alle Bitplanes der Bitmap
                                   allozieren  */
  for(i = 1; i <= Depth; i++)
   if((BMap->Planes[i-1]=AllocMem((Width*Height),
                    MEMF_CHIP|MEMF_CLEAR)) == NULL)
    exit(FALSE);               /* Fehler beim allozieren
                                    einer Bitplane  */
  return(BMap);
 }
```

```
FreeBitMap(BMap)        /*  Gibt den Speicher einer */
                        /*  Bitmap samt Bitplanes   */
                        /*  frei                    */

struct BitMap *BMap;

{
  int i;

  if(BMap == NULL)   exit(FALSE);

  /*  Bitplanes freigeben  */
  for(i = 0; i < (BMap->Depth); i++)
   FreeMem(BMap->Planes[i],(BMap->BytesPerRow)*
                                          (BMap->Rows));

  /*  BitMap Struktur freigeben  */
  FreeMem(BMap,sizeof(struct BitMap));

}


MakeDPFScr (x, y, w, h, Name, d, flags, font,
                                    Rast1, Rast2)

struct RastPort *(*Rast1), *(*Rast2);
short           x, y, w, h, d;
LONG            flags, font;
char            *Name;

{
 struct Screen   *Screen;
 struct BitMap   *BitMap1, *BitMap2;
 short           i,j, depth;
```

```
 depth = d / 2+d % 2;
 printf(" depth %d  \n",depth);

 /*  Bitmaps für die Playfields erzeugen    */
 BitMap1 = MakeBitMap(depth,w,h);
 BitMap2 = MakeBitMap(d/2,w,h);

Screen = (struct Screen *)MakeScr(x,y,w,h,Name,depth,
           flags | DUALPF,font,BitMap1);
 if(Screen == NULL)       /*  Fehler ?  */
  exit(FALSE);

 /*  Und die Bitplanes des zweiten Playfields
                                    eintragen  */
 /*  RastPort-Struktur initialiseren und die
                              neue Bitmap eintragen  */

 *Rast2 = AllocMem(sizeof(struct RastPort),NULL);
 if (*Rast2 == NULL)
  exit(FALSE);
 InitRastPort((*Rast2));

 (*Rast2)->BitMap = BitMap2;
 *Rast1 = &(Screen->RastPort);
 Screen->ViewPort.RasInfo->Next->BitMap = BitMap2;
 ScrollVPort(&(Screen->ViewPort));

 return(Screen);
}

CloseDPFScreen(Screen)
struct Screen *Screen;

{
FreeBitMap((Screen->ViewPort).RasInfo->Next->BitMap);
  /*FreeBitMap((Screen->ViewPort).RasInfo->BitMap);*/
  CloseScreen(Screen);
}
```

Kapitel 9

===============
Der Blitter
===============

Inzwischen ist der Amiga nicht mehr der einzige erschwingliche Computer mit einem 68xxx Prozessor, grafischer Fensteroberfläche und sehr guten Grafikmöglichkeiten. Was ihn aber nach wie vor von der Konkurenz abhebt sind seine Sonderchips, von denen der Blitter (aus dem Englischen Block Image Transferrer) mit der wichtigste ist. Er ist für das flexible und schnelle Kopieren und Füllen von Speicherbereichen zuständig und somit für Grafikanwendungen unentbehrlich. Ohne Blitter würden viele Operationen, die mit der Bildschirmausgabe zu tun haben, um Potenzen langsamer sein. Wir wollen Sie in diesem Kapitel aber nicht nur mit den Möglichkeiten des Blitters, sondern auch mit seiner Bedienung bekannt machen. Dabei werden Sie am Anfang die Blitterroutinen des Systems und anschließend die direkte Hardwareprogrammierung kennenlernen. Bei dieser werden wir uns im wesentlichen auf die Bereiche beschränken, die über Systemprozeduren nicht zugänglich sind.

1. Die Möglichkeiten des Blitters

Einfach nur zu sagen der Blitter diene zum Kopieren, ist eigentlich eine grobe Untertreibung, auch wenn dies eine wichtige Anwendung ist. Das gute Stück kann nämlich eine ganze Menge mehr als nur stupide Bits von einer Stelle zur anderen zu schaufeln. Wir wollen an dieser Stelle eine kurze Übersicht über seine Fähigkeiten geben:

(1) Extrem schnelles kopieren (über 15 Millionen Bits pro Sekunde) zwischen mehreren Speicherbereichen bis zur einer Größe von 1024x1024.

(2) Die Quell (source)- und Zielbereiche können durch eine beliebige logische Funktion verknüpft werden. Die logische Verknüpfung bezieht sich auf die Bits der beiden Speicherbereiche. So könnte man bestimmen, daß beim Kopieren nur dort im Zielbereich Bits gesetzt werden, wo vorher weder in der Quelle noch im Ziel etwas gesetzt war.

(3) Unterstützung von "rechteckigen Bereichen", d.h., daß die zu bearbeitenden Bereiche in Zeilen und Spalten eingeteilt werden können. Diese Eigenschaft ist natürlich für die Grafikprogrammierung besonderes nützlich.

(4) Füllen beliebiger Flächen mit vorgegebenen Füllmustern.

(5) Zeichnen von Linien mit beliebigen Strichmustern.

(6) Bitweises "shiften" von Speicher- oder Bildbereichen.

Die meisten dieser Möglichkeiten werden bereits so gut von der Systemsoftware unterstützt, daß man ohne direkten Hardwarezugriff auskommt. Dabei wird auch die Zusammenarbeit mit anderen Elementen des Grafiksystems wie Bitplanes und Rastports, sowie die Einbindung in das Multitasking-System berücksichtigt.

Bekanntlich ist Nichts perfekt und so hat auch der Blitter eine wichtige Einschränkung. Die Speicherauschnitte, die er bearbeiten kann müssen im CHIP-RAM liegen. Ein Versuch eine Adresse im FAST-RAM für eine Blitteroperation anzugeben würde im Überschreiben eines unbestimmten Bereiches des CHIP-RAMs resultieren.

2. Logische Verknüpfungen von Bereichen

Ein nicht ganz banales Problem beim Kopieren ist die Verknüpfung der Quelle(n) mit dem Ziel, die durch den sogenannten **Minterm** bestimmt wird. Theoretisch existieren 256 Möglichkeiten das endgültige Aussehen des Ziels festzulegen, wobei eine logische Verknüpfung zwischen den Quellbereichen und dem Zielbereich, sowie Shiftoperationen stattfinden können. Neben den raffinierten Verknüpfungen ist natürlich auch ein einfaches Kopieren oder Invertieren möglich. Um eine dieser Möglichkeiten auszuwählen muß zuerst der dazugehörende Wert des Minterms bestimmt und an den Blitter oder eine entsprechende Prozedur übergeben werden. Wir wollen hier zuerst die Shiftoperationen weglassen, da sie von den Systemprozeduren nicht direkt unterstützt werden. Ein Kopiervorgang mit dem Blitter (kurz: ein **Blitt**) kann immer als eine Verknüpfung von bis zu drei Bereichen zu einem Vierten betrachtet werden. Dabei wird eine Kombination der drei logischen Grundoperationen

AND (und)
OR (oder)
NOT (Negation)

auf die korrespondierenden Bits der Quellen angewandt und das Ergebnis in das entsprechende Bit des Ziels hineingeschrieben. Zur Demonstration betrachten wir die drei folgenden drei Bit breiten Quellen A, B und C:

A = 001
B = 110
C = 010

aus denen das Ziel D als

D = A OR (B AND C) hervorgeht.

Das Ergebnis ist dann also:
Bit 0 = 0 OR (1 AND 0) = 0
Bit 1 = 0 OR (1 AND 1) = 1
Bit 2 = 1 OR (0 AND 0) = 1

und somit:

D = 011

Bei den Blitteroperationen, die von Systemprozeduren unterstützt werden, können immer nur zwei Bitplanes berücksichtigt werden. Dabei wird die Quellbitmap mit der Zielbitmap verknüpft und das Ergebnis in die Zielbitmap geschrieben.

Wie wird die gewünschte Verknüpfung nun mit Hilfe des Minterms ausgewählt? Es ist jeder Dreierkombination von Bitwerten der drei Quellbereiche A, B und C ein Wert folgendermaßen zugeordnet:

A	B	C	Wert
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	4
1	0	0	8
1	0	1	16
1	1	0	32
1	1	1	64

Möchte man, daß immer beim Auftreten einer bestimmten dieser Kombinationen im Zielbereich das entsprechende Bit gesetzt wird, dann gibt man dem Minterm den aus der Tabelle abzulesenden Wert. Man kann natürlich auch festlegen, daß es mehrere Möglichkeiten gibt, die zum Setzen des Bits führen. In diesem Fall wird einfach die Summe aus den entsprechenden Werten als Minterm benutzt. Wer mit formaler Logik nicht vertraut ist, wird sich vieleicht fragen, was diese Wertetabelle mit logischen Verknüpfungen zu tun hat. Angenommen man möchte, daß im Ziel D ein Bit gesetzt wird, wenn die Bits der drei Quellen A, B und C an den dazugehörenden Stellen bestimmte Werte haben, so läßt sich dies als AND-Verknüpfung dieser Bits darstellen.

Die erste Zeile der Tabelle liest man sinnvollerweise so:

D = NOT A AND NOT B AND NOT C

oder kurz

D = ~A*~B*~C (NOT wird durch ~, AND durch * ersetzt)

Durch OR-Verknüpfungen solcher Terme kann festgelegt werden, daß nicht nur eine, sondern mehrere Zeilen der Tabelle zum Setzen des Bits führen können. Mit solchen OR-Verknüpfungen der in der Tabelle aufgeführten Werte kann dann auch jede beliebige andere Verknüpfung der Bereiche dargestellt werden. Die Bestimmung des Minterms erfolgt also in folgenden 3 Schritten:

(1) Die gewünschte logische Verknüpfung finden.

(2) Diese als OR-Verknüpfung der Kombinatinen aus der Tabelle darstellen.

(3) Die gefundenen Werte aus der Tabelle aufsummieren.

Um Ihnen die beiden letzten Schritte (der zweite kann recht schwierig sein) zu ersparen, folgt eine Tabelle, mit einigen wichtigen Verknüpfungen und die dazugehörenden Minterme. Wir haben der Übersichtlichkeit halber das NOT überall durch ~, das AND durch * und das OR durch + ersetzt.

Funktion	*Wert*
D = A	*$f0*
D = ~A	*$0f*
D = B	*$cc*
D = ~B	*$33*
D = C	*$aa*
D = ~C	*$55*
*D = A * C*	*$a0*
*D = A * ~C*	*$50*
*D = ~A * C*	*$0a*
*D = ~A * ~C*	*$05*
D = A + B	*$fc*
D = ~A + B	*$cf*
D = A + C	*$fa*
D = ~A + C	*$af*
D = B + C	*$ee*
D = ~B + C	*$bb*
*D = A * B*	*$c0*
*D = A * ~B*	*$30*
*D = ~A * B*	*$0c*
*D = ~A * ~B*	*$03*
*D = B * C*	*$88*
*D = B * ~C*	*$44*
*D = ~B * C*	*$22*
*D = ~B * ~C*	*$11*

D = A + ~B	*$f3*
D = ~A + ~B	*$3f*
D = A + ~C	*$f5*
D = ~A + ~C	*$5f*
D = B + ~C	*$dd*
D = ~B + ~C	*$77*
*D = A * B + ~A * C*	*$ca*

3. Einfaches Kopieren zwischen zwei Rastports

Eine der Grundanwendungen des Blitters ist das Kopieren von Bildauschnitten zwischen zwei oder auch innerhalb eines Rastports. Dies wird z.B. beim Verschieben, Vergrößern und Verkleinern von Fenstern oder bei der Ausgabe von Gadgets benötigt. Die Graphics-Library beinhaltet eine Prozedur namens **ClipBlit**, mit deren Hilfe Rastport-Abschnitte besonders einfach kopiert werden können. Die ***ClipBlit***-Prozedur, die die einfachste Blitter Kopierprozedur ist, braucht folgende Eingaben:

(1) Die Adresse der ***RastPort***-Datenstrukturen der beiden betroffene Rastports. Dabei darf der Quellrastport nach Bedarf mit dem Zielrastport identisch sein.

(2) Die Koordinaten der linken oberen Ecke des zu kopierenden Ausschnittes innerhalb des Quellrastports.

(3) Die Koordinaten innerhalb des Zielrastports, an der die linke oberen Ecke des Ausschnittes kopiert werden soll.

(4) Die Breite und die Höhe des zu kopierenden Ausschnittes.

(5) Ein ***MinTerm***, der die logische Verknüpfung der Ziel- und Quellbereiche angibt. Der Minterm wird bestimmt wie im vorigen Abschnitt beschrieben.

Die ***ClipBlitt***-Routine kann wie folgt aufgerufen werden, um den Bereich von der Position ***(x1,y1)*** der Breite ***dx*** und der Höhe ***dy*** aus dem Rastport ***Rast1*** in den Rastport ***Rast2*** an die Position ***(x2,y2)*** zu kopieren:

ClipBlit(Rast1,x1,y1,Rast2,x2,y2,dx,dy,192);

Für den Fall, daß die Positionsangaben oder die Größe nicht mit den Maßen des Rastports übereinstimmen, wird die entsprechende Anpassung (Clipping) von der Routine automatisch vorgenommen, so daß kein Absturz passieren kann/sollte. Auch die Bestimmung der Bitplanes, die von der Aktion betroffen werden, erfolgt automatisch.Die Anwendung der ***ClipBlit***-Prozedur zeigt eingehend das nachfolgende Beispielprogramm. Es macht auf dem Workbench-Screen ein Fenster auf und zeichnet dort zwei Rechtecke, eins mit der Farbe 1 und eins mit der Farbe 3; Diese werden dann anschließend samt der rechts von Ihnen liegenden leeren Bereichen mit verschiedenen Minterms nach unten kopiert. Dabei werden, um die Auswirkung der logischen Verknüpfungen zu zeigen, die kopierten Rechtecke teilweise übereinander kopiert.

Programm 9.1 BlittDemo

```
#include"Display.h"
#include"graphics/gfxbase.h"
#include"intuition/intuition.h"

struct Window    *Window;
struct RastPort  *Rast;

main ()
{
 USHORT  code, i, j, h;
 ULONG   Class;

 OpenIntui();
 OpenGfx();


 /*  Ein Fenster auf dem Workbench-Screen öffnen   */
 Window = (struct Window *)MakeWindow(100,20,400,200,
          0,0,"BlittDemo", WINDOWCLOSE|ACTIVATE,
          CLOSEWINDOW,NULL);
 if(Window == NULL)    /*  Fehler beim öffnen ?  */
  exit(FALSE);

 /*  Adresse des Viewports und des Rastports holen */
 Rast = Window->RPort;

 /*  Vierecke zeichnen   */
 SetAPen(Rast,1);
 RectFill(Rast,100,10,200,30);

 SetAPen(Rast,3); RectFill(Rast,125,15,175,25);

 /*  Rechteck 1 mittels Blitter nach Unten kopieren
                                      (MinTerm = 192) */
 ClipBlit(Rast,100,10,Rast,100,40,200,20,192);
```

```
/* Rechteck 1 mittels Blitter nach Unten inverse
   kopieren (MinTerm = 48 ) */
ClipBlit(Rast,100,10,Rast,100,70,200,20,48);

/* Rechteck 1 mittels Blitter nach Unten inverse
   kopieren (MinTerm = 48) */
ClipBlit(Rast,100,10,Rast,100,100,200,20,48);

/* Rechtecke 1 und 3 mittels Blitter mit 3 OR
   verknüpfen (MinTerm = 224) */
ClipBlit(Rast,100,10,Rast,100,100,200,20,224);

SetAPen(Rast,1);
RectFill(Rast,150,130,250,155);

/* Rechteck 1 mittels Blitter drüber NAND kopieren
   (MinTerm = 112) */
ClipBlit(Rast,100,10,Rast,100,130,200,20,112);

SetAPen(Rast,1);
RectFill(Rast,150,160,250,185);

/* Rechteck 1 mittels Blitter drüber NOR kopieren
   (MinTerm = 16) */
ClipBlit(Rast,100,10,Rast,100,160,200,20,16);

Class = WaitEvent(Window,&code);
CloseWindow(Window);

}
```

4. Weitere Kopiermöglichkeiten

Nachdem im vorigen Abschnitt ein einfaches Beispiel der Blitteranwendung gezeigt wurde, werden Sie nun die ganze Palette der Blitterkopierroutinen kennenlernen. Hierbei handelt es sich um Prozeduren, die sich auf Bitmaps beziehen. Die durch die Graphics-Library zur Verfügung gestellten Routinen unterstützen nicht nur das Kopieren von einzelnen Bitplanes sondern auch der gesammten Bitmap, wobei die betroffenen Bitplanes selbstverständlich frei ausgewählt werden können. Die einfachste dieser Prozeduren heißt ***BltBitMap***.

Zu den bei **ClipBlit** beschriebenen Parametern kommen bei ihr noch zwei weitere dazu:

(1) Eine Maske, die die betroffenen Bitplanes angibt.
(2) Falls innerhalb einer einzigen Bitmap gearbeitet wird, kann zur Beschleunigung ein Zwischenpuffer angegeben werden.

Logischerweise werden anstelle der Rastport-Adressen an gleicher Stelle die Adressen der entsprechenden **BitMap** - Datenstrukturen übergeben. Die Maske, die die zu kopierende Bitplanes bestimmt, ist ein Byte in dem jedes Bit stellvertretend für eine Bitplane ist. Ist dieses Bit gesetzt, so wird diese Bitplane von der Kopieroperation betroffen, ist es gelöscht so wird sie ausgelassen. Falls Sie also nur die Planes 1 und 3 (oder deren Auschnitte) transferieren wollen, dann müssen Sie als Maske den Wert ***2 Hoch 1 + 2 Hoch 3 = 2 + 8 = 10*** benutzen.

Um den Bereich von der Position **(x1,y1)** der Breite **dx** und der Höhe **dy** aus der Bitmap **Bit1** in die Bitmap **Bit2** an die Position **(x2,y2)** zu kopieren genügt der folgende Aufruf:

BltBitMap(Bit1,x1,y1,Bit2,x2,y2,dx,dy,192,Mask,NULL);

Die **NULL** als letzter Parameter bedeutet, daß kein Puffer zur Verfügung gestellt wird.

Es ist auch möglich, Bildaten aus einer Bitmap in einen Rastport zu transferieren. Dazu gibt es die ***BltBitMapRastPort***-Prozedur. Sie wird ähnlich wie ***ClipBlit*** aufgerufen. Der Unterschied besteht darin, daß der erste Parameter ein Adreßzeiger auf eine Bitmap ist ,auf die sich auch die ersten beiden Parameter beziehen. Die Angabe der zu kopierenden Bitplanes kann mit Hilfe des ***Mask***-Feldes der ***RastPort***-Datenstruktur erfolgen. Für den dort stehenden Wert gilt das gleiche wie für den ***Mask*** Parameter von ***BltBitMap***. Durch die Beinflussung dieses Feldes kann übrigens auch die Auswahl der Bitmaps bei Verwendung der ***ClipBlit***-Prozedur erfolgen.

Die letzte Prozedur, die wir in diesem Abschnitt vorstellen wollen ist eigentlich nur eine Abwandlungen von ***BltBitMapRastPort***. Sie heißt ***BltMaskBitMap-RastPort*** und besitzt einen zusätzlichen Parameter, der einen Adreßzeiger auf eine "Schablonenbitmap" ist. Diese Schablone muß die Maße des Ziels haben und bestimmt welche Punkte durch den Blitt beinflußt werden. Alle Punkte die in der Schablonenbitmap nicht gesetzt sind, wirken bei der Blitteroperation als Sperren, d.h. die korrespondierenden Punkte des Ziels werden von der Blitteroperation nicht betroffen. Als Beispiel betrachten wir wieder zwei "Minibitmaps" A

und B, sowie eine Schablonenbitmap C mit je einer Bitplane, je einer Zeile und 3 Spalten. Wenn sie vor der Operation so ausehen:

A = 110
B = 001
C = 010

dann wird nach dem Blitt in B

011

stehen. Es wurde also nur das Bit 1, das in der Schablonenbitplane gesetzt ist kopiert. Die Bits 0 und 2 blieben unverändert, da die Schablone dort "undurchsichtig" ist. In dem nachfolgenden Programm finden Sie alle der oben besprochenen Prozeduren wieder. Es öffnet zwei Fenster auf dem Workbench-Screen und zeichnet in das obere, kleinere ein Viereck der Farbe 3. Dieser wird dann mit verschiedenen Blitterprozeduren (***BltBitMapRastPort***, ***BltBitMap*** und ***BltMaskBitMapRastPort***) und verschiedener Bitplanesauswahl mehrmals in das untere Fenster kopiert. Vor der Anwendung der ***BltMaskBitMapRastPort***- Prozedur wird zunächst die "Schablonenbitmapplane" mit der drunter definierten ***MakeMaskPlane***-Prozedur erzeugt und mit horizontalen Streifen gefüllt. Daher wird dann auch das kopierte Rechteck gestreift erscheinen.

Programm 9.2 BitMapBlitt

```
#include"exec/memory.h"
#include"Display.h"
#include"graphics/gfxbase.h"
#include"intuition/intuition.h"
```

```
/*  Koordinaten der linken oberen Ecke des ersten
                                             Fensters  */
#define Winx 200
#define Winy 10

/*  Koordinaten der linken oberen Ecke des zweiten
                                             Fensters  */
#define Win2x 100
#define Win2y 60

struct Window     *Window, *Window2;
struct RastPort   *Rast, *Rast2;
struct BitMap     *Bit, *Bit2;

char *Mask;

LONG Len;

main (){
 USHORT  code, i, j, h;
 ULONG   Class;

 OpenIntui();
 OpenGfx();


 /*  Ein Fenster auf dem Workbench-Screen öffnen  */
 Window = (struct Window *)MakeWindow(Winx,Winy,239,
          40,0,0,"Window 1",WINDOWCLOSE|ACTIVATE,
          CLOSEWINDOW,NULL);

 if(Window == NULL)    /*  Fehler beim Öffnen ?  */
  exit(FALSE);

 /*  Adresse des Rastports und der Bitmap holen  */
 Rast = Window->RPort;
 Bit = Rast->BitMap;
```

```
Window2 = (struct Window *)MakeWindow(Win2x,Win2y,
          439,190,0,0,"Window 2",ACTIVATE,
          CLOSEWINDOW,NULL);

if(Window2 == NULL)    /* Fehler beim Öffnen ?  */
 exit(FALSE);

/*  Adresse des Rastports und der Bitmap holen   */
Rast2 = Window2->RPort;
Bit2  = Rast2->BitMap;

/*  Viereck im ersten Fenster zeichnen   */
SetAPen(Rast,3);
RectFill(Rast,70,15,180,35);

/* Aus der Bitmap des ersten Fensters in den
                   Rastport des zweiten blitten  */
BltBitMapRastPort(Bit2,70+Winx,10+Winy,Rast2,50,
                  20,100,20,192);

Rast2->Mask = 1;
BltBitMapRastPort(Bit2,70+Winx,10+Winy,Rast2,160,
                  20,100,20,192);

Rast2->Mask = 2;
BltBitMapRastPort(Bit2,70+Winx,10+Winy,Rast2,270,
                  20,100,20,192);

/* Aus der Bitmap des ersten Fensters in die Bitmap
   des zweiten blitten  */
BltBitMap(Bit,70+Winx,10+Winy,Bit2,50+Win2x,60+Win
          2y,100,20,192,3,NULL);

BltBitMap(Bit,70+Winx,10+Winy,Bit2,160+Win2x,60+Win
          2y,100,20,192,1,NULL);

BltBitMap(Bit,70+Winx,10+Winy,Bit2,270+Win2x,60+Win
          2y,100,20,192,2,NULL);
```

```
 /*  Aus der Bitmap des ersten Fensters in die Bitmap
                                  des zweiten blitten   */
 /*  Dabei wird eine Blittmaske benutzt   */
 Mask = MakeMaskPlane(Bit);
 Rast2->Mask = 3;
 BltMaskBitMapRastPort(Bit,70+Winx,10+Winy,Rast2,50,
                       100,100,20,192,Mask);
 Rast2->Mask = 1;
 BltMaskBitMapRastPort(Bit,70+Winx,10+Winy,Rast2,160,
                       100,100,20,192,Mask);
 Rast2->Mask = 2;
 BltMaskBitMapRastPort(Bit,70+Winx,10+Winy,Rast2,270,
                       100,100,20,192,Mask);

 FreeMem(Mask,(Bit->BytesPerRow) * (Bit->Rows));
 Class= WaitEvent(Window,&code); CloseWindow(Window);
 CloseWindow(Window2);
}

MakeMaskPlane(BMap)

/* Diese Prozedur erzeugt eine Schablonenbitplane */
/* Sie wird mit dünnen horizontalen Streifen
                                          gefüllt */
struct BitMap *BMap;
{
  char *Mem;
  LONG Len, End;

  Mem = NULL;
  Len = (BMap->BytesPerRow) * (BMap->Rows);
  Mem = AllocMem(Len,MEMF_CHIP|MEMF_CLEAR);
  if(Mem == NULL)
   exit(FALSE);
  for(End = Len + Mem; Mem < End; ++Mem)
  *(Mem++) = 12;
  return(Mem-Len);
}
```

5. Noch mehr Blitterroutinen

Neben den reinen Kopierroutinen, die bis jetzt vorgestellt wurden, gibt es in der Graphics-Library noch drei weitere nütztliche Prozeduren. Dies sind:

(1) BltClear
(2) BltPattern
(3) BltTemplate

Die erste dient zum Löschen (also mit Nullen füllen) eines Speicherbereiches. Sie wird als:

BltClear(MemPtr,Bytes,Flags)

aufgerufen, wobei ***MemPtr*** ein Adreßzeiger auf den Speicherbereich und ***Bytes*** die Größe des zu löschenden Bereiches ist. Der dritte Parameter, ***Flags*** gibt weitere Informationen zum Löschmodus an. Bit 1 gibt hier an, ob das Programm gestoppt werden soll, bis die Operation beendet ist (Bit gesetzt) oder nicht (Bit gelöscht). Das zweite Bit entscheidet darüber, ob der Speicherbereich als "normaler" Speicher (Bit gelöscht), oder als eine Bitplane (Bit gesetzt) interpretiert werden soll. Im ersten Fall wird ***Bytes*** ganz normal als Anzahl der zu löschenden Bytes interpretiert und der Bereich wird gelöscht. Im zweiten Fall werden allerdings die oberen 16 Bits von Bytes als die Anzahl der zu löschenden Zeilen, und die unteren als die Anzahl der zu löschenden Bytes pro Zeile angesehen. So können beliebige Bildausschnitte sehr einfach gelöscht werden.

Die zweite Prozedur, ***BltPattern*** ist eine Abwandlung der ***BltMaskBitMapRastPort***-Prozedur. Sie hat das Aufrufformat:

BltPattern(Rast,Mask,x1,y1,x2,y2,Bytes)

Dabei wird der Bereich des Rastports ***Rast*** mit der linken oberen Ecke bei ***(x1,y1)*** und der rechten unteren Ecke bei ***(x2,y2)*** mit dem Muster gefüllt, auf das ***Mask*** zeigt. Das Muster wird in Form einer "Schablonenbitplane" gespeichert. Diese muß mindestens genauso groß wie das zu füllende Rechteck sein. Seine Breite in Bytes wird in ***Bytes*** angegeben. Um z.B. ein Rechteck der Breite 10 Pixel zu füllen, muß in ***Bytes*** 2 angegeben werde, was bedeutet, daß die Schablone 2 Byte breit ist. Schablonen dürfen nur ganze Vielfache von 8, also 8, 16, 24 etc. Pixel breit sein und müssen selbstverständlich in CHIP-RAM liegen.

Die letzte der Routinen, die wir in diesem Abschnitt beschreiben wollen, ***BltTemplate***, dient dazu, in "gepackten Arrays" gespeicherte Daten auszulesen. Solche "gepackten Arrays" haben Sie schon im Zusammenhang mit der Speicherung der Bilddaten der Fonts im Kapitel 7 kennengelernt. Zur Errinnerung:

Es werden Daten für einzelne Bilder zeilenweise (also alle erste Zeilen hintereinander, dann alle zweiten Zeilen hintereinander usw.) gespeichert, wobei jedes Zeichen eine andere Breite in Punkten haben kann. So können also das eine mal in einem Wort drei Daten einer Bildzeile von 2 Zeichen haben (wenn z.B. jedes 8 Bit breit ist) und das andere mal von 2.5 Zeichen (z.B. 2 a 6 Bit und die Hälfte von einem 8 Bit breitem). Um jetzt ein in so gestalteten Fontbilddaten gespeichertes Zeichen der Breite ***Width*** und

der Höhe ***Height*** an der Position ***(x,y)*** des Rastports ***Rast*** auszugeben muß ***BltTemplate*** wie folgt aufgerufen werden:

BltTemplate(Source,BitPos,Mod,Rast,x,y,Width,Height);

In ***Source*** muß die Anfangsadresse der Bildaten stehen. Der Parameter ***BitPos*** gibt die Anfangsposition des auszugebenden Zeichens innerhalb der Arrays in Bits an.

Die Anzahl der Bytes, die überspringen werden müssen um von einer Zeile zur anderen zu gelangen, müssen Sie in ***Mod*** angeben.

6. Der Blitter und das Multitasking

Der Amiga ist bekanntlich ein Multitaskingrechner, es laufen also gleichzeitig mehrere Programme ab. Das bedeutet auch, daß es durchaus möglich ist, daß mehrere Tasks gleichzeitig auf die Idee kommen, den Blitter zu benutzen. Da dieser aber nur sequentiell arbeitet, gibt es im Betriebssystem Mechanismen, die solche Konflikte beseitigen und einem Programm den alleinigen Zugriff auf den Blitter sichern. Die einfachste Alternative ist hier die Anwendung der Parameterlosen ***WaitBlit-***, ***OwnBlitter-*** und ***DisownBlitter-***Prozeduren. Die erste dient dazu, solange zu warten, bis der Blitter frei ist. Der Aufruf der ***OwnBlitter-***Routine bewirkt, daß das aufrufende Programm den Blitter in Besitz nimmt und bis zur Freigabe uneingeschränkt benutzten kann.

Die Freigabe erfolgt mit ***DisownBlitt***. Es ist für den Amiga lebenswichtig, nach Beendigung des Blitterzugriffs diesen wieder freizugeben, da ohne Blitter die gesamte Bildschirmausgabe und einige andere Funktionen nicht ablaufen können. Vor dem Aufruf der Systemprozeduren, die bis jetzt vorgestellt wurden, braucht man sich normalerweise den Blitter nicht noch extra mittels ***OwnBlitter*** zu sichern, da diese dies intern erledigen. Will man aber, daß eine Reihe von Blitterbefehlen ohne zeitliche Verzögerung hintereinander ausgeführt wird, dann muß die gesamte Sequenz derart aufgebaut sein:

```
WaitBlit();
OwnBlitter();
<Blitterroutinen>
DisownBlitter();
```

Dadurch wird verhindert, daß zwischen den Aufrufen der einzelnen Blitterroutinen ihr Programm den Blitter verliert. Unentbehrlich ist die Beschaffung des Blitters natürlich bei den direkten Hardwarezugriffen, wie sie in späteren Abschnitten vorgestellt werden.

Bei der Hardwareprogrammierung bedient man sich auch oft einer anderen Methode, den Blitter für sein Programm zu sichern. Sie hat den Vorteil, daß solche Anfragen Vorrang vor ***OwnBlitter*** haben und außerdem eine Synchronisierung des Vorgangs mit dem Elektronenstrahl, der das Bild auf dem Monitor zeichnet, möglich ist.

So kann man z.B. den Bildschirmspeicher modifizieren, während der Strahl sich außerhalb des Bildschirms befindet. Die Grundlage dieses Verfahrens ist die Datenstruktur ***bltnode***. Im System existiert eine Liste solcher Strukturen. Jede von ihnen beschreibt einen sogenannten "Blitterjob", der nichts weiter als eine Liste von Blitteroperationen ist. Die Blitterjobs werden in der Reihenfolge der ***bltnode***-Strukturen in der Liste abgearbeitet. Durch einen Aufruf von ***QBlit*** können Sie eine solche Struktur, die Ihren eigenen Blitterjob beinhaltet, an das Ende dieser Liste einfügen. Bei der Initialisierung von ***bltnode*** muß folgendes getan werden:

(1) Die Adresse des Codes der Funktion, die die Blitteroperationen ausführt, muß in das Feld function eingetragen werden. Dieser Code sollte eigentlich so in Assembler geschrieben sein, daß er sowohl im User als auch in Superwisermodus (Prozessormodi des MC68000) läuft. Der Code wird solange aufgerufen und der Blitter nicht freigegeben,bis die Funktion in dem Datenregister D0 des MC68000 eine 0 zurückgibt. Da "C" bei einem ***return***-Aufruf das

Ergebnis ebenfalls in D0 zurückgibt, kann notfalls auch eine "C"-Funktion verwendet werden.

(2) Die Adresse einer "Cleanup"-Routine, die nach der letzten Blitteroperation ausgeführt wird, oder ***NULL*** wird in das ***cleanup***-Feld geschrieben. So ist es möglich am Ende des Blitterjobs, ihn wieder in die Jobschleife einzufügen.

(3) Je nachdem ob eine "Cleanup"-Routine aufgerufen werden soll oder nicht, muß der Wert ***CLEANUP*** ($040) oder 0 in das ***stat***-Feld hineingeschrieben werden.

Beim Aufruf von ***QBlitt*** wird als einziger Parameter die Adresse einer so initialisierten Struktur übergeben. Wenn Sie wollen, daß mit der Ausführung ihres Blitterjobs gewartet wird, bis eine bestimmte Position des Elektronenstrahl auf dem Bildschirm erreicht wird, dann müssen Sie sich der ***QBSBlit*** Prozedur bedienen. Sie wird genauso wie ***QBlitt*** benutzt. Sie müssen allerdings bei der Initialisierung der ***bltnode***-Struktur das ***beamsync*** Feld mit der Nummer der Zeile belegen, bei deren Durchlauf ihr Blitterjob ausgeführt werden soll. Auf Grund des Multitaskings (dieses Liste ist global für alle Tasks) kann es hierbei allerdings zu zeitlichen Konflikten kommen. Es kann nämlich sein, daß ein anderer Benutzer den Blitter solange behält, daß Ihre Position verpaßt wird.

Es ist wichtig, daß die Funktion, deren Adresse im ***function***-Feld von ***bltnode*** eingetragen wird, keine der Blitterroutinen der Graphics-Library aufruft. Diese Prozeduren warten nämlich intern auf den Blitter, den sie allerdings nicht bekommen können, weil ihn ihre Funktion besitzt. Ein Beispiel für die Anwendung der ***bltnode***-Strukturen finden Sie im nächsten Abschnitt.

7. Die Blitterhardware

Die Blitterroutinen sind zwar sehr flexibel und bequem, um jedoch den Blitter optimal auszunutzen, wird man ihm direkt durch Registerzugriffe ansteuern. Dies ermöglicht ein Arbeiten ohne auf die Graphics- oder Intuition-Library zugreifen zu müssen, wie man es z.B. bei der Programmierung eines Vorspanns machen muß. Wir werden deswegen in diesem Abschnitt den Hardwareaufbau des Blitters und seine Funktionsweise kurz beschreiben. Der Blitter ist kein eigenständiger Chip. Er ist in den Sonderchip AGNUS des Amiga integriert. Daher liegen auch seine Register, wie auch alle Register der Sonderchips, bei $dff000 plus einen registerspezifischen Offset. Um sie von "C" aus anzusprechen benutzt man am besten die ***Custom***-Datenstruktur. Die Komponenten dieser Datenstruktur entsprechen sowohl in der Reihenfolge, als auch in ihrer jeweiligen Länge den Registern der Customchips. Wenn man also eine solche Struktur an der Adresse $dff000 vereinbart, dann kann man einfach über ihre Komponenten auf die Register zugreifen, ohne sich um die Offsets Gedanken machen zu müssen. Sie werden diese Funktionsweise in dem Beispielprogramm des nächsten Abschnitts sehen.

Nun zu den Registern des Blitters. Er besitzt acht 16-Bit Adreßregister, die paarweise zu 4 32-Bit Adreßregistern organisiert sind, vier Moduloregister (je eins zu einem Adreßregister), zwei Controllregister, vier Datenregister, zwei Maskenregister und ein Größenregister, das gleichzeitig zur Aktivierung eines Blitts dient. Um eine Blitteroperation durchzuführen, muß folgendes getan werden:

(1) Die Adressen der drei Quellen und des Ziels müssen in die Adressregister eingetragen werden. Dies sind die Regsiter:

BLTAPTH Quelle A Hi
BLTAPTL Quelle A Lo

BLTBPTH Quelle B Hi
BLTBPTL Quelle B Lo

BLTCPTH Quelle C Hi
BLTCPTL Quelle C Lo
BLTDPTH Ziel Hi
BLTDPTL Ziel Lo

Wie Sie sehen, müssen Sie die Adressen in High- und Lowteil aufteilen. Dies ergäbe bei der Adresse $ffff0000

Hi = $ffff
Lo = $0000

(2) Die Modulowerte für die 3 Quellen und das Ziel in die Moduloregister holen:

BLTAMOD Modulowert für Quelle A
BLTBMOD Modulowert für Quelle B
BLTCMOD Modulowert für Quelle C
BLTDMOD Modulowert für Ziel

Der Modulowert beinhaltet die Anzahl der Worte in einer Zeile. Er wird nach dem Bearbeiten einer Zeile zu der Adresse addiert, um mit der nächsten fortfahren zu können.

(3) Durch Setzen der Bits 0 bis 7 des Controlregisters BLTCON0 die gewünschte logische Verknüpfung wählen. Sie müssen diese Bits einfach auf einen

entsprechenden Wert setzen, wie bereits in Abschnitt 2 besprochen.

(4) Die restlichen Bits des Controllregisters BLTCON0 initialiseren. Es sind die Bits 8 bis 11 für das Einschalten der DMA-Kanäle für die Quellen A bis C und des Ziels zuständig, sowie die Bits 12 bis 15 für die Verschiebung der Quelle A (siehe Anhang E).

(5) Durch entsprechedes initialiseren der Bits 0 bis 4 des BLTCON1 Controllregisters den Modus auswählen. Zum "normalen" kopieren müssen diese Bits alle gelöscht sein (siehe Anhang E).

(6) Die restlichen Bits des Controllregisters BLTCON0 initialiseren. Es sind die Bits 12 bis 15 für die Verschiebung der Quelle A zuständig (siehe Anhang E). Die Bits 5 bis 11 haben keine Bedeutung.

(7) Die Größe des zu bearbeitenden Bereiches in das BLTSIZE-Register schreiben. Die unteren 6 Bits (Bits 0 bis 5) geben die Breite in Wörtern (Ein Wort = 2 Bytes = 16 Bit), die oberen 10 (Bits 6 bis 15) die Höhe in Zeilen an. Eine Höhe von 0 wird dabei als 1024 interpetiert. Der Wert dieses Registers muß also (Höhe AND $3ff)*64 + (Breite $3f) sein. Das BLTSIZE-Register muß als letztes beschrieben werden, da ein Schreibzugriff auf dieses Register die Blitteroperation startet. Wir haben in diesem Abschnitt nur vom Kopieren von Zeilen gesprochen. Wenn einen nicht in Zeilen eingeteilter Bereich des Speichers kopiert werden soll, dann müssen Sie die Modulowerte einfach auf 1 setzen.

Ein einfaches Beispiel für die hardwaremäßige Blitterprogrammierung finden Sie im nachfolgenden Bei-

spielprogramm. Es bindet mittels QBlitt einen einfachen Kopierjob in die Jobliste ein.

```
#include"hardware/custom.h"
#include"hardware/blit.h"
#include"Display.h"
#include"graphics/gfxbase.h"
#include"intuition/intuition.h"

#define Winx 100
#define Winy 20

struct Window    *Window;
struct RastPort  *Rast;
struct BitMap    *BitMap;

struct Custom *Custom;

struct bltnode myNode;

long Blitt();


main ()
{
 USHORT  code, i, j, h;
 ULONG   Class;

 OpenIntui();
 OpenGfx();


 /*  Ein Fenster auf dem Workbench-Screen öffnen  */
 Window = (struct Window *)MakeWindow(Winx,Winy,400,
          200,0,0,"BlittDemo",WINDOWCLOSE|ACTIVATE,
          CLOSEWINDOW,NULL);
```

```
 if(Window == NULL)    /*  Fehler beim Öffnen ?  */
  exit(FALSE);

 /*  Adresse des Viewports und des Rastports holen */
 Rast = Window->RPort;
 BitMap = Rast->BitMap;

 /* Anfangsadresse der Customchips  laden */
 Custom = 0xdff000;

 myNode.function  = Blitt;
 myNode.cleanup   = NULL;
 myNode.stat      = 0;

 QBlit(&myNode);

  /*  Alles schließen  */
 Class = WaitEvent(Window,&code);
 CloseWindow(Window);

}


long Blitt()

{
  Custom->bltalwm = 0xffff;
  Custom->bltafwm = 0xffff;
  Custom->bltamod = BitMap->BytesPerRow;
  Custom->bltbmod = BitMap->BytesPerRow;
  Custom->bltcmod = BitMap->BytesPerRow;
  Custom->bltdmod = BitMap->BytesPerRow;
  Custom->bltapt  = BitMap->Planes[0];
  Custom->bltbpt  = BitMap->Planes[0];
         /*+ Winy *(640/16) + Winx/16; */
  Custom->bltcpt  = BitMap->Planes[0];
  Custom->bltdpt  = BitMap->Planes[1];
  Custom->bltcon0 = 192+SRCA+SRCB+DEST;
```

```
  Custom->bltcon1 = 0;
  Custom->bltsize = 266;
  while(Custom->dmacon && 1>>14);
  return(0);
}

int Blitt()

{
  BlittLine();
  return(0);
}

int fuck(b)

short b;
{  short a;

  a = b;
  return(0);

}
```

8. Fortgeschrittene Optionen beim Kopieren

Bis jetzt wurde beschrieben, wie eine Standardkopieroperation durchgeführt werden kann. Ein großer Nachteil bei dieser Operation besteht darin, daß nur ganze Wörter kopiert werden können. Nun weiß man aber von den Blitterroutinen der Graphics-Library, daß es möglich ist, Blitteroperationen auf Bereiche anzuwenden, die weder an Wortgrenzen liegen, noch eine durch 16 teilbare Breite haben. Dies wird durch zwei noch

nicht erklärte Fähigkeiten des Blitters ermöglicht: Shiften und Maskieren. Schon im vorigen Abschnitt wurde gesagt, daß die oberen Bits der beiden Controllregister die Verschiebung der Quelle A bzw. B in Bits angeben. Das heißt, daß jedes Wort, das aus der entsprechenden Quelle geholt wird, um die in diesen Bits angegebene Anzahl von Stellen (0 bis 15) nach rechts verschoben wird. Dabei werden die Bits,die bei einem Wort nach rechts "hinausgeschoben" werden, an die durch Verschieben freiwerdenden Anfangsstellen des nächsten Wortes der Zeile geschrieben. Dadurch kann eine gesamte Zeile geschiftet werden. In die freigewordenen Bits des ersten Wortes einer Zeile werden Nullen hineingeschiftet. Eine aus den zwei folgenden Worten:

1111000011111111 0101010101000000

bestehende Zeile wird nach einem Shift um 3 Bits wie folgt ausehen:

0001111000011111 1110101010101000

Auf diese Art kann ein Bild an eine Stelle kopiert werden, die nicht an einer Wortgrenze liegt. Wenn Sie einen Auschnitt an eine Position, die 4 Bits rechts von einer Wortgrenze liegt, kopieren wollen, dann brauchen Sie die Quelle(n) nur um diese 4 Bits zu verschieben. Dazu müssen Sie zu dem entsprechenden Controllregister (BLTCON0 für Quelle A, BLTCON1 für Quelle B) die Zahl

4 * 2 Hoch 11

addieren. Ein Problem bei dieser Methode wird deutlich wenn man daran denkt, daß beim Shiften um n Bits am Anfang einer Zeile n Nullen dazukommen, während die letzten n Bits verlorengehen. Man kann die letz-

ten Bits selbstverständlich retten, indem man ein Wort mehr kopiert, dabei kopiert man aber einige unerwünschte Bits mit.

Die Lösung dieses Problem bietet das Maskieren. Hinter diesem zunächst mysteriös anmutenden Wort verbirgt sich die Möglichkeit, für das erste und das letzte Wort einer Zeile 1 - Wort Masken anzugeben, die bestimmen welche Bits kopiert werden und welche nicht. Somit kann man die beim Shiften entstandenen Nullen im ersten Wort und die durch Kopieren eines zusätzlichen Wortes dazugekommenen unerwünschten Bits beseitigen. Die Maske für das erste Wort wird in das BLTAFWM-, die für das letzte in das BLTALWM-Register hineingeschrieben.

Es werden dann nur die Bits des ersten bzw. letzten Wortes kopiert, die in der Maske gesetzt sind. Um also hineingeschobene Nullen am Anfang zu beseitigen, muß man nur die *n* ersten Bits im BLTAFWM-Register auf Null, die restlichen auf 1 setzen. Analog können die uerwünschten Bits in dem letzen Wort durch Setzen der letzten Bits von BLTALWM auf Null ausgeblendet werden.

Das Maskieren kann übrigens auch benutzt werden, um ein Auschnitt, der ausserhalb einer Wortgrenze anfängt bzw. aufhört, zu kopieren. Man rundet den Bereich einfach auf Wortgrenzen auf und blendet das unerwünschte durch Maskieren aus. Leider kann eine Maske nur für die Quelle A angegeben werden. Es können aber trotzdem alle Operationen mit Maskieren durchgeführt werden. Notfalls muß man halt mehrmals kopieren.

Ein Problem über das bis jetzt noch nicht gesprochen wurde, ist das Kopieren von sich überlappenden Bereichen. Solange Sie abwärts kopieren also z.B die

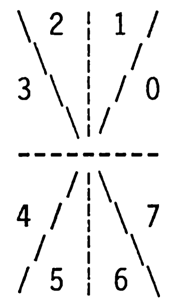
Zeilen 12 bis 17 einer Bitplane in die Zeilen 10 bis 15 der gleiche Bitplane übertragen wollen, läuft alles reibungslos. Zuerst wird die Zeile 12 in die Zeile 10, dann die 13 in die 11, dann 14 in 12 usw. kopiert. Wenn Sie allerdings probieren umgekehrt vorzugehen, gibt es Schwierigkeiten. Bei Kopieren der Zeilen 10 bis 15 in die Zeilen 12 bis 17 wird zuerst die Zeile 10 in die Zeile 12 dann die 11 in die 13 usw. kopiert. Dabei wird aber der Inhalt der Zeilen 12 bis 15 zerstört, bevor er umkopiert werden kann! Um auch solche Situationen zu meistern, kann durch Setzen des Bits 1 des BLTCON1-Registers die Richtung des Kopiervorgangs umgekehrt werden (dieser Modus wird **Descending-Mode** genannt). Wenn Sie dieses Bit setzen, dann müssen Sie statt der Anfangsadresse der Quellen und des Ziels die Endadressen angeben. Der Blitter kopiert dann von Hinten, in unserem Beispiel also zuerst die Zeile 15 in die Zeile 17, dann 14 in 16 dann 13 in 15 usw.

9. Zeichnen von Linien

Eine Fähigkeit des Blitters, die die Blitterroutinen des Systems nicht offenbaren, ist das Zeichnen von Linien. Durch Setzen des LINE-Bits (Bit 1) des Controllregisters BLTCON1 kann der Linienmodus eingeschaltet werden. Die Angabe der gewünschten Anfangs- und Endkoordinaten der Linie ist leider nicht ganz einfach. Der Blitter beschreibt eine Linie, die vom Punkt *(x1,y1)* zum Punkt *(x2,y2)* geht, durch folgende Parameter:

(1) Oktant:

Dieser Parameter ordnet die Linie auf Grund ihrer Richtung einem der acht gleicher Kreisteile zu. Die Einteilung des Kreises in Oktanten sieht in etwa so aus:



Die Zahlen geben die Oktantennummern an. Um den richtigen Oktanten für Ihre Linie zu finden, können Sie sich der nachfolgenden Tabelle bedienen. Neben der Linienkoordinaten werden dort die Werte ***dx*** und ***dy*** benutzt,die die Differenzen der x- bzw. y-Koordinaten der Anfangs- und Endpunkte sind: Sie werden als

dx = abs(x1 - x2)
dy = abs(y1 - y2)

berechnet.

Oktant	Voraussetzungen
0	x1<=x2; y1<=y2; dx>=dy
1	x1<=x2; y1<=y2; dx<=dy
2	x1>=x2; y1<=y2; dx<=dy
3	x1>=x2; y1<=y2; dx>=dy
4	x1>=x2; y1>=y2; dx>=dy
5	x1>=x2; y1>=y2; dx<=dy
6	x1<=x2; y1>=y2; dx<=dy
7	x1<=x2; y1>=y2; dx>=dy

(2) Die Steigung

Zur Berechnung der Steigung werden wieder die bei der Oktantenbestimmung berechneten Werte ***dx*** und ***dy*** benötigt. Zuerst müssen der kleinere und der größere der beiden Werte bestimmt werden. Wir bezeichnen den kleineren mit ***kd*** und den größeren mit ***gd***. Es ist also

kd = minimum(dx,dy)
gd = maximum(dx,dy)

Mit diesen Bezeichnern wird die Steigung durch drei Ausdrucke festgelegt:

(1) 2*kd
(2) 2*kd-gd
(3) 2*kd -2*gd

Zusätzlich muß noch geprüft werden ob ***2*kd*** größer als gd ist

(3) Die Anfangsadresse des Startpunktes

Gemeint ist hier die Adresse im Bildspeicher, an der das Wort mit dem ersten Punkt der Linie liegt. Wenn die Bitplane, in der die Linie gezeichnet wird, in ***Plane*** gespeichert ist und die Bitplane ***Lines*** Zeilen a ***Bytes*** Bytes hat, dann berechnet sich die gesuchte Anfangsadresse durch:

Plane + (Lines -y1-1)*Bytes + 2*(x1/16)

(4) Die Position des Startpunktes im Wort

Wenn der Anfangspunkt der Linie nicht an einer Wortgrenze liegt, muß noch der Abstand von dem Wort-

anfang angegeben werden. Wenn die x-Koordinate z.B. bei 17 liegt, dann beträgt der Abstand 1, da bei 16 ein neues Wort anfängt.

Nachdem Sie nun den Verlauf der Linie auf diese Art Blittergerecht formuliert haben, können Sie die Blitteroperation anlaufen lassen. Die Blitterregister müssen dazu folgendermaßen initialisiert werden:

(1) Den Wert $8000 in BLTADAT schreiben.

(2) Die Maske für das Zeichnen der Linie in BLTBDAT schreiben. Für eine durchgehende Linie ist dies $ffff.

(3) Die Register BLTAFWM und BLTALWM auf $ffff setzen.

(4) Die Moduloregister folgendermaßen initialiseren:

BLTAMOD = 2*kd - 2*gd
BLTBMOD = 2*kd
BLTCMOD = Die Breite der gesamten Bitplane in Bytes
BLTDMOD = Die Breite der gesamten Bitplane in Bytes

(5) In die Quellen- und das Zielregister folgendes schreiben:

BLTAPT = 2*kd - gd
BLTBPT = unbenutzt
BLTCPT = Die Adresse des ersten Pixels der Linie.
BLTDPT = Die Adresse des ersten Pixels der Linie.

(6) In die Bits 15 bis 12 des BLTCON0-Registers die Position des Anfangspunktes innerhalb des Wortes (also x1 modulo 15) schreiben.

(7) Bits 8,9,11 des BLTCON0-Regsiters auf 1, Bit 10 auf 0 setzen.

(8) Den Wert $CA für die logische Verknüpfung AB+AC in die unteren 8 Bits (Bits 0 bis 7) von BLTCON0 schreiben.

(9) Die Bits des BLTCON1 Registers folgendermaßen setzen:

Bits 12 bis 15: Hier die Stelle angeben, an der das in BLTBDAT bestimmte Muster in der Linie anfangen soll, angeben (normalerweise 0).

Bits 7 bis 11: Unbenutzt.

Bit 6 = 1 falls 2*kd gd, = 0 sonst.

Bit 5 = 0 (unbenutzt)

Bits 2 bis 4 = Diese Bits geben die Nummer des Quadranten an. Jedem Quadranten wird ein Wert zugeordnet, den diese Bits annehmen müssen. Hier die Zuordnungstabelle:

Oktant	*Wert*
0	*6*
1	*1*
2	*3*
3	*7*
4	*5*
5	*2*
6	*0*
7	*4*

Bit 1 = 1;

Bit 0 = 1 (Linienmodus einschalten)

(10) Die unteren 6 Bits (Bits 0 bis 5) von BLTSIZE auf 2, die oberen (Bit 6 bis 15) auf ***gd+1*** setzen.

Auch beim Zeichnen von Linien ist darauf zu achten, daß das BLTSIZE-Register, daß die Operation startet, als letztes beschrieben wird.

10. Füllen von Flächen

Die letzte noch nicht besprochenen Anwendung des Blitters ist das Füllen von Flächen. Das Füllen kann ohne Zeitverlust in den Kopiervorgang integriert werden. Die Daten, die aus den drei Quellen gemäß der angegebenen Verknüpfungsvorschrift erzeugt und in BLTDDAT geschrieben werden, werden vor der Übertragung in den Zielbereich entsprechend behandelt. Zum Füllen braucht der Blitter eine durch 1 - Bit breite durchgehende Linien seitlich bergrenzte Fläche. Diese könnte beispielsweise diese Form haben:

0010000000000001000
0010000100010000100 0
0010000000000001000

Die Fülloperation wird durch setzen des EFE (Bit 4) oder des IFE-Bits (Bit 3) des BLTCON1 Registers eingeschaltet. Alle anderen Einstellungen gleichen den beim Kopieren.Je nach dem welches der beiden Bits gesetzt ist, werden die Begrenzungsbits mit ins Bild übernommen (IFE = Inclusive Fill Enable) oder weggelassen (EFE = Exclusive Fill Enable). Im EFE Modus würde die Beispielfläche nach dem Füllen so aussehen:

00011111111111110000
00011110000011110000
00011111111111110000

Wenn man dagegen den IFE-Modus benutzt, dann bleiben die Begrenzungen bestehen. Das Ergebnis sieht dann so aus:

00111111111111111000
00111111000111111000
00111111111111111000

Falls man man möchte, daß nicht innerhalb der Begrenzung, sondern ausserhalb gefüllt wird, dann kann muß man zusätzlich noch das FCI (Fill Carry In) Bit (Bit 2) des BLTCON1-Register setzen. Im IFE-Modus ergibt sich dann für die Beispielfläche nach dem Füllen:

11100000000000001111
11100001111100001111
11100000000000001111

Eine Besonderheit des Füllens liegt darin, daß er nur im Descending-Mode (also Bit 1 von BLTCON1 = 1 siehe Abschnitt 8) funktioniert.

Am Anfang dieses Abschnitts wurde bereits gesagt, daß das Füllen während einer Kopieroperation erledigt wird. Wenn Sie also einen Bereich Füllen wollen, dann müssen Sie ihn auf sich selbst kopieren und dabei den gewünschten Füllmodus einschalten.

KAPITEL 10

==============

Der Copper

==============

In diesem Kapitel wollen wir den zweiten Graphikcoprozessor des Amiga vorstellen. Während der Blitter diverse Dienstleistungen übernimmt, ohne die eine schnelle Graphikdarstellung nicht denkbar wäre, ist der Copper für die Koordinierung des Aufbaus des Displays zuständigt. Er ist dafür verantwortlich, daß zum richtigen Zeitpunkt (der Zeitpunkt wird durch die Position des Rasterelektronenstrahls bestimmt) die richtigen Werte in den richtigen Regsitern stehen, so daß an jeder Bildschirmposition die Daten angezeigt werden, die auch dorthin gehören. Alle Angaben, die bezüglich der Anordnung und Art der gerade angezeigten Viewports gemacht werden, werden vom System in Listen von Copperanweisungen übersetzt. Somit braucht sich der Hauptprozessor um die Bildausgabe nicht zu kümmern und ist frei für andere Aufgaben.

1. Die Funktionen des Coppers

Der Copper ist ein eigenständiger Prozessor, der Programme in ihm verständlicher Sprache ausführen kann. Allerdings ist er in seinem Befehlsatz sehr beschränkt. Er verfügt lediglich über drei Befehle: ***WAIT***, ***MOVE*** und ***SKIP***, sowie sieben Register.

Außerdem ist der Adreßbereich, auf den der Copper zugreifen kann, auf die Register der Customchips beschränkt. So wenig dies ist, für den Zweck den der Copper zu erfüllen hat reicht es völlig aus. Die Beeinflussung der Custom-Chipregister ist alles was nötig ist, um das Ausgabebild zu steuern. Zusätzlich kann der Copper den Blitter benutzen (seine Register liegen auf den Customchips) und somit indirekt auf den gesamten CHIP-Speicher zugreifen.

Es ist weiterhin möglich, daß der Copper einen Interrupt beim 68000 auslöst und so den CPU in seine Dienste stellt.

Das Copperprogramm befindet sich in der sogenannten Copperliste. Sie ist ein Speicherbereich in dem Copperanweiungen aneinandergereiht sind und dessen Adresse in einem der Copperregister befindet.

Nun zu den drei Copperbefehlen. Sie haben folgende Funktionen:

(1) MOVE
Dieser Befehl erlaubt es einen bestimmten Wert in eines der Register der Customchips zu schreiben. Normalerweise sind die Register $00 bis $20 vom Copperzugriff ausgeschlossen. Durch Setzen eines bestimmten Bits in einem der Copperregister sind für ihn aber auch die Register $10 bis $20 erreichbar.

(2) WAIT
Der WAIT-Befehl wartet darauf, daß der Elektronenstrahl eine bestimmte Position erreicht hat.

(3) SKIP
Ein SKIP-Befehl veranlaßt den Copper dazu, den nachfolgenden Befehl zu überspringen, wenn die aktuelle Position des Strahls größer oder gleich ist, als die bei dem Befehl angegebenen Werte. Mit seiner Hilfe läßt sich durch Manipulation der Copperregister (Der Copper kann ja auch auf seine eigenen Register zugreifen) eine bedingte Verzweigung realisieren.

Die Copperlisten bestehen normalerweise aus einer Abfolge von Blöcken der Form

WAIT
<Mehrere Move-Anweisungen>

Sinnvollerweise sollten diese Anweisungsblöcke in der Reihenfolge der bei den *WAIT*-Befehlen angegebenen Elektronenstrahlpositionen geordnet sein. Wäre dies nicht der Fall, dann könnte es sein, daß bestimmte Anweisungen nie zur Ausführung kommen.

2. Der hardwaremäßige Aufbau des Displays

Aus vorigen Kapiteln ist Ihnen bekannt, daß die Bildanzeige aus Viewports besteht, von denen jeder eine andere Auflösung, Farbzusammensetzung, und Größe haben und seine Bilddaten in einer eigenen Bitmap lagern kann.Diese Viewports werden zu einer View (siehe Kapitel 8) zusammengefaßt. Nun stellt sich die Frage, wie aus diesen getrennt gespeicherten Bausteinen das Bild des Gesamtdisplays entsteht und welche Rolle dabei der Copper spielt. Um diesen Vorgang zu verstehen, muß man zuerst etwas über die Grafikdarstellung auf Hardwareebene wissen. Die Erzeugung des Bildsignals aus Daten die in einem Speicherbereich liegen, wird bekanntlich von den Customchips übernommen. Diese müssen aber die Speicheradressen kennen, wo sich die darzustellenden Daten befinden (der Amiga verfügt nicht wie einige andere Computer über einen festen Bildschirmspeicher) und wie sie dargestellt werden sollen, also welche Farben, Auflösung etc. benutzt werden sollen. Zu diesem Zweck verfügen sie über eine Gruppe von Registern, die große Ähnlichkeit mit eini-

gen aus vorherigen Kapiteln bekannten Strukturen der Graphics-Library haben. Der wesentliche Unterschied besteht darin, daß jede Veränderung dieser Register auch eine sofortige Veränderung des Ausgabebildes bewirkt. Die in den Strukturen der Graphics- und der Intuition-Librarary gespeicherten Daten gewinnen erst dadurch Bedeutung, daß Sie im richtigen Augenblick in die entsprechenden Register übertragen werden. Wir wollen hier einige wichtige auflisten und beschreiben. Wir geben dabei für die Assemblerprogrammierung die relative Lage der Register zu $dff000 (siehe Kapitel 9)und für "C" den Namen der entsprechenden Komponente der Custom-Datenstruktur. Diese Datenstruktur muß selbstverständlich wie im Kapitel 9 beschrieben, an der Speicherstelle $dff000 liegen, damit ein Zugriff möglich wird. Wenn ein "x" im Namen steht, dann bedeutet dies, daß es mehrere solche Register gibt, die an dieser Position durchnummeriert sind.

(1) Die Bitplane Pointer Register (BPLxPT)

Diese Register entsprechen den Adreßzeigern auf die einzelnen Bitplanes, die sich in jedem ***BitMap*** Datensatz befinden. Sie beinhalten die Adresse der Bitplanes, die momentan angezeigt werden. Es gibt sechs BPLxPT-Register, so wie es maximal sechs Bitplanes geben kann. Sie liegen ab $E0 im Speicher, können nur beschrieben werden und sind je vier Bytes Lang (zwei Bytes für Adresse High-Byte und zwei Byte für Adresse LowByte). Von "C" aus sind sie als ***Custom.bpltp[x]*** (x = Nummer der Plane 1 bis 6) ansprechbar.

(2) Die Playfield Controlregister (BPLCONx)

Es gibt drei dieser Register (0 bis 2). Sie sind je zwei Byte lang und liegen bei $100. Von "C" sind sie

durch ***Custom.bplcon0***, ***Custom.bplcon1*** und ***Custom.-bplcon2*** zu erreichen. Besonderes interessant sind folgende Bits des BPLCON0-Registers:

Bit 15 - Schaltet den HiRes-Modus ein.
Bits 12 bis 14 - Geben die Anzahl der benutzen Bitplanes an.
Bit 11 - Schaltet den HAM-Modus ein.
Bit 10 - Schaltet den Dual-Playfield-Modus ein.
Bit 2 - Schaltet den Interlace-Modus ein. Dieses Register entspricht also dem ***ViewMode***-Feld der ***ViewPort***-Datenstruktur.

(3) Die Color Register (COLORxx)

Diese Register entsprechen der zu jedem Viewport gehörenden ***ColorMap***. Jedes dieser 32 Register (0 bis 31) ist 16 Bit lang und beinhaltet die RGB Zusammensetzung der momentan angezeigten Farben. Diese Register liegen bei $180. In der ***Custom*** Struktur sind sie als ***color[32]*** definiert.

Den direkten Zugriff auf eines dieser Register, das BPLPT1-Register demonstriert das nachfolgende Programm. Es schreibt den Adreßzeiger auf die Bitplane 0 des aktiven Screen in dieses Register. Dadurch wird der Hardware vorgetäuscht, daß die Bitplanes 0 und 1 des Screens identisch seien, und alles auf dem Screen erscheint in der Farbe 3.

Programm 10.1 CustomPoke

```
#include"exec/types.h"
#include"hardware/Custom.h"
#include"Display.h"
```

```
struct Custom *Custom;
APTR  Plane;

main()
{
  long i;
  OpenIntui();

  /*  Zeiger auf Bitplane 0 des aktiven Screens
                                          holen  */
Plane= IntuitionBase->ActiveScreen->BitMap.Planes[0];

  Custom = 0xdff000;           /*  Basisadresse der
                                   Customchips       */
  for(i = 0; i < 100000; i++)

   /* Zeiger auf die Bitplane 0 des aktiven Screens
      als Hardwarebitplane 1 */

   Custom->bplpt[1] = Plane;
}
```

Was ist nun hierbei die Aufgabe des Coppers? Nun, sobald der Elektronenstrahl eine Position erreicht, an der ein neuer Viewport (z.B. ein Screen der zur Hälfte nach unten geschoben wurde) anfängt, erreicht wird, schreibt der Copper die entsprechenden Werte in die Customchip-Register. Dazu gehören unter anderem die Adreßzeiger auf die Bitplanes mit den Bilddaten und der Inhalt der Farbregister. Beim nächsten Durchlauf des Strahls werden dann wieder die Werte des oben liegenden Vieports eingetragen.

Ein letztes nützliches Register das wir hier erwähnen wollen, ist das INTREG-Register (liegt bei $9c von "C" als Custom->intereq ansprechbar). Durch Setzen des vierten Bits dieses Registers kann der Copper einen Prozessorinterrupt auslösen.

3. Copperlisten und Views

Wie bereits gesagt, werden sämtliche Informationen, die das Aussehen einer View betreffen, also die Anordnung und Form der Viewports, sowie eventuell vorhandene Sprites, in Form von Copperlisten kodiert. Jeder Viewport besitzt in der ***ViewPort***-Datenstruktur in den Komponenten ***DspIns***, ***SprIns*** und ***ClrIns*** Adreßzeiger auf ***CopList***-Strukturen. Jede solche Datenstruktur beschreibt eine eigene Copperliste. Die erste (in ***DspIns*** für **Disp**lay **Ins**tructions) beinhaltet Befehle, die zum Aufbau der Anzeige des Viewports benötigt werden. Diese Copperliste wird beim ***MakeVPort***-Befehl aus den Daten in den ***ViewPort***- und ***RasInfo***-Datenstrukturen erzeugt. Aus diesem Grund bewirken nur Veränderungen in einer der beiden Strukturen ohne einen ***MakeVPort***-Aufruf keine Änderung des Ausgabebildes. Die beiden anderen werden für Sprites benötigt. Zusätztlich gibt es in der ***UCopIns***-Komponente einen Adreßzeiger auf eine ***UCopList***-Datenstruktur, die die sogenannte User Copperliste beinhaltet. Mit Hilfe dieser Datenstruktur kann eine weitere, benutzerdefinierte Copperliste in die Viewportliste eingefügt werden. Alle diese Copperlisten sind wie verlangt nach Elektronenstrahlpositionen geordnet.

Da die Viewports nur als Teile einer View angezeigt werden können, reicht es nicht, daß jeder Viewport seine eigenen Copperlisten hat. Die Einzelnen Listen müssen, bevor sie die Bildausgabe beinflussen können, zu einer weiteren geordneten Einheit verbunden werden.

Diese Aufgabe erledigt der ***MrgCop***-Befehl. Er arbeitet sich durch alle Viewports einer View durch und

verknüpft dann geordnet alle ihre Copperlisten. In der *View*-Struktur selbst gibt es auch zwei Felder, in die eine Copperliste eingetragen wird: ***LofCpreList*** und ***ShfCprList***. In dem ersten Feld befindet sich immer ein Zeiger auf eine Copperliste. Das zweite Feld wird nur im Interlace-Modus benötigt und enthält die Copperliste, die beim zweiten Durchlauf, also für die geraden Zeilen, benötigt wird.

Der letzte Schritt bei der Erzeugung des Ausgabebildes ist das Eintragen der Copperliste der gerade aktiven View in das entsprechende Hardwareregister des Coppers. Diese Aufgabe wird von der ***LoadView***-Prozedur erledigt.

4. Erstellen und Anbinden einer neuen Copperliste

Nun sind Sie in der Lage das im Kapitel 8 beschriebene Verfahren zur Erstellung einer eigenen View und auch einige andere Dinge, die mit der Bildausgabe zusammenhängen, genau zu verstehen. Der nächste Schritt besteht darin, aktiv in das Geschehen einzugreifen und die Anzeige mit Hilfe eines eigenen Programms zu verändern. Die beste Möglichkeit hierzu besteht darin, eine eigene Copperliste zu erstellen und sie an die aktive View anzubinden. Den Zeiger auf die aktive View holt man sich auf die bekannte Art über die ***GfxBase***-Datenstruktur. Die Anbindung erfolgt einfach durch Eintragen einer ***UCopList***-Datenstruktur in das ***UCopIns***-Feld eines der Viewports dieser View. Diese Datenstruktur muß selbstverständlich vorher mit der gewünschten Copperliste initialisiert werden, denn es macht wenig Sinn eine leere Copperliste anzubinden. Die Initialiserung erfolgt in folgenden Schritten:

(1) Zuerst muß die Datenstruktur selbst erzeugt werden. Sie können sie nicht einfach als Variable vereinbaren, denn sie muß im CHIP-RAM liegen. Die Allozierung kann so ausehen:

```
CopList = AllocMem(sizeof(struct UCopList),MEMF_CHIP
                   |MEMF_CLEAR|MEMF_PUBLIC);
```

(2) Die Befehlssequenz muß in die Struktur eingetragen werden. Zu diesem Zweck gibt es in "graphics/gfxmacros.h" folgende zwei Makros: *CWAIT* und *C-MOVE*. Das erste dient dazu, den *WAIT*-Befehl des Coppers in die Liste einzutragen. Als Parameter werden ihm die Adresse der User-Copperliste übergeben, sowie die Zeile und Spalte, auf die gewartet werden soll. Um also in die Liste ***List*** den *WAIT*-Befehl auf Zeile 10 Spalte 0 einzutragen, genügt folgender Aufruf:

```
CWAIT(List,10,0);
```

Dem zweiten Makro, das einen *MOVE*-Befehl der Liste hinzufügt, wird ebenfalls als erstes Argument die Adresse der User-Copperliste übergeben. Die beiden anderen Argumente geben dann die Adresse des betroffenen Customchipregisters und den einzutragenden 16 Bit Wert an. Der Befehl

```
CMOVE(List,Custom->color[1],0)
```

bewirkt das Eintragen des *MOVE*-Befehls, der eine Null in das Colorregister 0 schreibt, in die UserCopperliste ***List***.

(3) Die Copperliste muß mit Hilfe des *CEND*-Makros abgeschlossen werden. Es wird mit der Adresse der Liste als einziger Parameter aufgerufen.

Nach dem Eintragen der fertigen Liste in den Viewport müssen nur noch die Prozeduren ***MakeVPort***, ***MrgCop***, und ***LoadView*** in dieser Reihenfolge aufgerufen werden, um die Änderung zur Anzeige zu bringen.

In dem nachfolgenden Beispielprogramm haben wir auf die soeben beschriebene Weise eine Copperliste, die den von diversen PD-Demos bekannten "Rainbow"-Effekt erzeugt,an die aktive View herangehängt. Sie bewirkt, daß zwischen den Zeilen 10 und 200 nach jeweils 10 Zeilen der Hintegrund eine um eine RGB-Stufe dunkleren Blauwert annimmt. Es werden also auf dem Workbenchscreen, der eine Tiefe von zwei hat, 20 Farben gleichzeitig angezeigt werden. Es wird dazu einfach in der neuen User-Copperliste mit der Zeile 20 beginnend auf jede nächste durch zehn Teilbare Zeile mit ***WAIT*** gewartet und dann der neue Farbenwert in das ***COLOR00***-Hardwareregister eingetragen. Die entsprechende Copperliste kann einfach in einer **for**-Schleife erstellt werden.

Damit Sie nach dem Starten dieses Programms auch wieder die normale Anzeige bekommen, wird nach kurzen Warten die neue Copperliste entfernt.

Programm 10.2 Stripes

```
#include"exec/types.h"
#include"exec/memory.h"
#include"hardware/Custom.h"
#include"Display.h"
#include"graphics/gfxmacros.h"
```

```
struct Custom *Custom;
struct View *View;
struct ViewPort *VPort;
struct UCopList *CopList, *oldCopList;
main()

{
  long i;

  OpenGfx();
  OpenIntui();

  /*  Zeiger auf die aktive View holen  */
  View = GfxBase->ActiView;

  /*  Zeiger auf den Viewport des aktiven Screens
                                            lesen  */
  VPort = &(IntuitionBase->ActiveScreen->ViewPort);

  /*  Basisadresse der Customchips  */
  Custom = 0xdff000;

  /*  Copperliste Allozieren  */

  CopList = AllocMem(sizeof(struct UCopList),MEMF
                      CHIP|MEMF_CLEAR|MEMF_PUBLIC);
  if (CopList == NULL)
    exit(FALSE);

  /*  Eigene Copperliste mit den Systemmakros
                                      erstellen  */
  for(i = 10; i <= 200; i += 10)
  {
   CWAIT(CopList,i,0);          /*  Warten bis der
                    Strahl zehn Zeilen weiter ist */
   CMOVE(CopList,Custom->color[0],i/10);
                 /*  Farbregister 0 verändern  */
  }
```

```
 CEND(CopList);    /*  Copperliste abschließen  */

/*  Copperliste in den Viewport einbinden  */

oldCopList = VPort->UCopIns;  /*  Alte User
                    Copperliste merken        */

VPort->UCopIns = CopList;     /*  Neue eintragen
                                               */

MakeVPort(View,VPort);        /*  Und in die
                     globalen Liste einbinden  */
MrgCop(View);
LoadView(View);


for(i = 0; i < 500000; i++);  /*  Abwarten  */

/*  Die Alte Copperliste wieder einbinden  */
VPort->UCopIns = oldCopList;

/*  Alte Kopperlist wieder eintragen        */
MakeVPort(View,VPort);

/*  Und in die globalen Liste einbinden  */
MrgCop(View);
LoadView(View);

FreeMem(CopList,sizeof(struct UCopList));

}
```

5. Die Copperhardware

Zum Schluß dieses Kapitels wollen wir für die, die direkt mit der Copperhardware arbeiten möchten, die Copperregister, sowie das dazugehörige Befehlsformat beschreiben. Der Copper besitzt die folgenden sieben Register:

COP1LCH, COP1LCL ($80 in "C" Custom->cop1lc)

Diese Register beinhalten die 18 Bit-Adresse der ersten Copperliste.

COP2LCH, COP2LCL ($84 in "C" Custom->cop2lc)

Diese Register beinhalten die 18 Bit-Adresse der zweiten Copperliste.

COPJMP1 ($88 in "C" Custom->copjmp1)

Durch Beschreiben dieses Registers wird der Copper veranlaßt, die in COP1LC1 stehende Adresse als die Adresse des nächsten auszuführenden Befehls zu übernehmen.

COPJMP2 ($88 in "C" Custom->copjmp2)

Durch Beschreiben dieses Registers wird der Copper veranlaßt, die in COP1LC2 stehende Adresse als die Adresse des nächsten auszuführenden Befehls zu übernehmen.

COPCON ($02e in "C" Custom->copcon)

Dies ist das Kontrollregister des Coppers. Es enthält nur ein einziges Bit (Bit 0). Dieses Bit gibt an, ob der Copper auch auf die Register $10 bis $1f zugreifen kann (Bit 0 gesetzt) oder nicht (Bit 0 gelöscht).

Wie der obigen Registerbeschreibung zu entnehmen ist, muß man, um eine neue Copperliste anzuhängen, ihre Adresse in das COP1LC-Register (unter diesem Namen fassen wir die COP1LCH- und COP1LCL-Register zusammen) schreiben und dann auf das COPJMP1-Register zugreifen, damit der Copper dieses Adresse übernimmt. Die COP2LC - und COPJMP2-Register werden für bedingte Sprünge mit Hilfe des ***Skip***-Befehls gebraucht. Die in COPJMP1 stehende Adresse wird jedes mal übernommen, wenn der Rasterstrahl die Position (0,0) erreicht.

Auf diese Art wird die Copperliste bei jedem Durchlauf des Elektronenstrahls aufs neue durchlaufen, so daß das gewünschte Anzeigebild permanent aufrechterhalten wird. Da es keinen Stopbefehl für den Copper gibt, muß am Ende einer jeden Liste ein ***WAIT*** auf eine unmögliche Strahlpostion stehen. Dadurch wird gewährleistet, daß an dieser Stelle gewartet wird, bis beim Erreichen der Position (0,0) die Abarbeitung der Copperliste von Vorne beginnt. Ein solcher Befehl ist z.B.,

WAIT($ff,$ff)

da es keine Horizontalen Positionen, die größer als $E2 sind, gibt. Alles was Sie jetzt noch wissen müssen, um eigene Copperlisten zu erstellen, ist das Befehlsformat. Alle drei Copperbefehle bestehen aus zwei Prozessorworten. Die Bits 0 der beiden Worte bestimmen folgendermaßen, um welchen Befehl es sich handelt:

Bit 0 Wort 1	Bit 0 Wort 2	Befehl
0	0	MOVE
0	1	MOVE
1	0	WAIT
1	1	SKIP

Wie Sie sehen wird der *MOVE*-Befehl allein durch das erste Wort bestimmt. Die Bedeutung der restlichen Bits der beiden Befehlsworte für die einzelnen Befehle können Sie der nachfolgenden Auflistung entnehmen:

MOVE

Wort 1
Bit 0 - Immer 1 (Befehlserkennung).
Bits 1 bis 8 - Die Adresse (Nummer) des Zielregisters.
Bits 9 bis 15 - Unbenutzt, auf 0 setzen.

Wort 2
Bits 0 bis 15 - Das Datenwort, das in das Zielregister geschrieben wird.

WAIT

Wort 1
Bit 0 - Immer 0 (Befehlserkennung)
Bits 1 bis 7 - Angabe der horizontalen Strahlposition.
Bits 8 bis 15 - Angabe der vertikalen Strahlposition.

Wort 2
Bit 0 - Immer 0 (Befehlserkennung)
Bits 1 bis 7 - Maske für die horizontale Strahlposi-

tion.
Bits 8 bis 14 - Maske für die vertikale Strahlposition.
Bit 15 - Blitter Finish Disable Bit

SKIP

Wort 1
Bit 0 - Immer 1 (Befehlserkennung)
Bits 1 bis 7 - Angabe der horizontalen Strahlposition
Bits 8 bis 15 - Angabe der vertikalen Strahlposition.

Wort 2
Bit 0 - Immer 1 (Befehlserkennung)
Bits 1 bis 7 - Maske für die horizontale Strahlposition.
Bits 8 bis 14 - Maske für die vertikale Strahlposition.
Bit 15 - Blitter Finish Disable Bit

Die Bedeutung der Maskenbits in den zweiten Befehlswörtern der Befehle ***WAIT*** und ***SKIP*** bedarf noch einer Erläuterung: Sie geben an, welche Bits in der Angabe der horizontalen bzw. vertikalen Strahlposition für den Vergleich mit den beim Befehl angegebenen Werten von Bedeutung sind. Es werden nur die Bits getestet, die in der Maske gesetzt sind. Für alle anderen wird der Test als ***TRUE*** angenommen. Setzt man z.B. sowohl die Maske der vertikalen Position als auch die Position selbst auf $08, dann trifft auf diese Angabe jede achte Position zu. Wenn man, wie es meistens der Fall ist, nur an einer bestimmten vertikalen Position interessiert ist, dann sollte man eine Null in der Maske für die horizontale Position angeben.

Zur Verdeutlichung des Umgangs mit den Befehlen betrachten Sie die folgende Copperliste:

WAIT($0f,0)
MOVE($180,$ffff)
WAIT(ff,fe)

Diese Liste bringt den Copper dazu, jeweils beim Erreichen der Zeile 50 in das Farbregister 0 die Farbe Weiss zu schreiben. Die letzte WAIT-Anweisung dient nur dem Abschluß der Liste durch Warten auf eine unmögliche Position. In Hexzahlen umgesetzt würde diese Liste so aussehen:

```
$0f01 $ff00 /* WAIT($0f,0)         */
$0180 $ffff /* MOVE($180,$ffff)    */
$ffff $fffe /*WAIT(ff,fe)          */
```

Zum Schluß noch ein Paar Worte zum *SKIP*-Befehl. Dazu betrachten Sie das Konstrukt der Form:

```
MOVE(COP1LC,Marke)Marke:
<Anweisungen>
SKIP(Position,Maske)
MOVE(COPJMP1,0)
MOVE(COP1LC,AlterWert)
<Rest der Liste>
```

Solange die in Skip angegebene Position nicht erreicht ist, erfolgt ein Schreibzugriff auf das COP-1LC-Register, wodurch der Copper zu der Adresse Marke verzweigt (Sie wurde am Anfang ja in COP1LC hineingeschrieben).

Dadurch werden die Anweisungen zwischen der Marke

und dem *SKIP*-Befehl in einer Schleife immer wieder ausgeführt. Erst wenn die angegebene Position überschritten wurde, wird der Zugriff auf COPJMP1 unterlassen. Es erfolgt also kein Rücksprung und die Abarbeitung der Liste wird sequentiell fortgesetzt.

Falls wie in diesem Beispiel der Wert in COP1LC während der Abarbeitung der Copperliste verändert wurde, dann ist unbedingt darauf zu achten, daß der alte Wert vor dem Erreichen des Endes der Copperliste wiederhergestellt wird. Dies ist notwendig, damit der Copper den Anfang der Liste wieder anspringen kann.

Kapitel 11

==

Techniken der zweidimesionalen Grafikerzeugung

==

In diesem Kapitel wollen wir Ihnen möglichst viele konkrete Beispiele der "einfachen" Grafikprogrammierung geben. Dabei kommt es uns nicht so sehr auf die Vollständigkeit der Behandlung der einzelnen Themen an, sondern wir möchten Ihnen vielmehr möglichst viele verschiedene Beispiele geben. Daher wird am Ende eines jeden Abschnittes eine Rubrik "Tips" stehen, in denen wir Ihnen noch zahlreiche Anregungen zu dem jeweiligen Thema geben werden. Nachdem wir Ihnen also die Grundzüge einer jeweiligen Grafikprogrammiertechnik vor Augen geführt haben, animieren die nachfolgenden Tips Sie hoffentlich dazu, selbst Hand an die Algorithmen zu legen. Dabei sind diese Tips wirklich nur als Anregung aufzufassen und sollten Ihre eigene Kreativität in keiner Weise beeinträchtigen.

1. Erste Schritte

Will man mit möglichst geringem Programmieraufwand bereits grafische Effekte auf dem Bildschirm erzeugen, so eignen sich dazu besonders die FOR-Schlaufen in Verbindung mit dem ***Line***, ***Polygon***, oder ***Circle***-Befehl. Bei dieser Technik liegt oft mehr Programmierarbeit in der "Definition" der Arbeitsumgebung, also öffnen eines Screens, eines Fensters und ganz wichtig: das Zuweisen der Farbregister. Gerade auf dem Amiga sollte man bei der Grafikprogrammierung, bis auf wenige Ausnahmen, sich mit weniger als 32 Farben nicht zufriedengeben. So stehen Ihnen also in diesem Fall 32 aus 4096 Möglichkeiten der Farbregisterbelegung zur Verfügung.

Doch gerade mit der Farbauswahl kann man den "Betrachtungswert" einer Grafik vervielfachen, so daß

es sich lohnt, wenn Sie sich dafür genügend Zeit nehmen.

Die einfachste Art eine Schlaufe mit einem Grafikbefehl zu belegen, ist das direkte Einsetzen der Schlaufenzählvariable in einen Grafikbefehl, so daß also zum Beispiel der Radius einer Folge von Kreisen verändert wird. Die so entstehenden Bilder erscheinen aber meißt schon sehr bald langweilig, da die Veränderungen ja grundsätzlich linear sind. Wesentlich interessanter wird es, wenn man stattdessen einen Funktionswert der Schlaufenzählvariable einsetzt.

Optisch geeignet sind hier besonders die Sinus- und die Kosinusfunktion, aber auch mit den Exponential-, Logarithmus- und Exponentialfunktionen, sowie deren beliebigen Kombination lassen sich abwechslungsreiche Effekte erzielen. Wir beschränken uns zunächst einmal auf den Sinus und Kosinus, da Sie gegenüber anderen Funktionen einen gewaltigen Vorteil haben: Ganz egal was für Werte man einsetzt, wie groß oder wie klein sie auch sein mögen, der Funktionswert liegt immer zwischen -1 und +1. Dies erspart uns Kopfzerbrechen darüber,mit was wir den erhaltenen Funktionswert multiplizieren müssen, damit ein Wert herauskommt, der auch im Gültigkeitsbereich unseres Fensters liegt. Gehen wir von einem Fenster der Breite 320 Pixels und der Höhe 230 Pixels (dann ist oben noch Platz für die Titelbalken des Screens und Fensters), so multiplizieren wir lediglich die Funktionswerte mit der halben Höhe, bzw. Breite und addieren anschließend den gleichen Wert noch einmal. Nach der Multiplikation erhalten wir also Werte zwischen -160 und +160 (-115 und +115), nach der nochmaligen Addition Werte zwischen 0 und 320 (0 und 230). Jetzt können wir loslegen und in einer Schlaufe Werte in die soeben neu definierten Funktionen einsetzen. Das Ergebnis können wir direkt als x- und y-Koordinate auf den Bildschirm

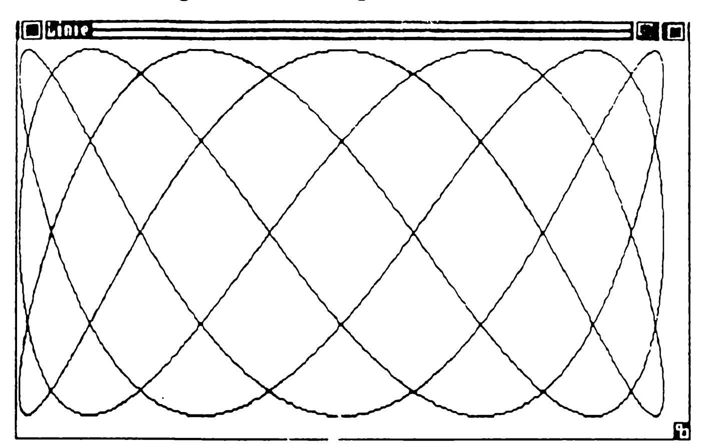
werfen. Eine sich über den Bildschirm schwingend fortbewegende Linie erhalten wir durch den folgenden Algorithmus:

```
Linie (a,b)
float a,b;
/* a im Bereich von 0.0 und 360.0 und b > a */

{
  fx = 75;          /* Beliebiger x-Schwingungsfaktor */
  fy = 200;         /* Beliebiger y-Schwingungsfaktor */
  Calculate(a); /* Initialisierung von x und y */
  for (n = a;      /* Schlaufe: von a */
       n < b;      /*            bis b */
       n+=1);      /* n inkrementieren */
      {Calculate (n); /* neues Koordinatenpaar
                                   berechnen */
       Line(xold,yold,x,y,farbe); /* Verbinden mit
                                 den alten Werten */
       xold = x; /* Zwischenspeichern der
                    aktuellen      */
       yold = y; /* x- und y-Werte */
  };
}

Calculate (i)
/* Berechnen der aktuellen Koordinaten */
floot i;
{
  x = sin(i*pi/fx) * 160.0 + 160.0;  /* Berechne
                                      x-Koordinate */
  y = cos(i*pi/fy) * 115.0 + 115.0;
                                     /* Berechne
                                      y-Koordinate */
}
```

Bild 11.1 - Ausgabe des Algorithmus "Linie".



Um ein funktionsfähiges Programm zu erhalten, muß natürlich die Arbeitsumgebung entsprechend gesetzt, vor allem aber ein (GZZ-)Fenster der Größe 320/230 geöffnet sein. Die so entstehende, schwingende Linie können Sie variieren, indem Sie den Schwingungsfaktoren einen Wert zwischen jeweils 30 und 270 zuordnen. In unserem Programm ***Linie***, daß Sie auf der Begleitdiskette finden, werden weiterhin auch den jeweiligen Kurvenabschnitten zyklisch die verschiedenen, gesetzten Farben zugewiesen.

Wir können die entstehenden Grafiken verschönern, indem wir der Linie einen Partner geben, der zwar ähnlich, aber doch völlig unabhängig schwingt. Dazu führen wir zwei weitere Schwingungsfaktoren ***fx1*** und ***fy1*** ein, sowie 2 weitere xy-Koordinatenpaare. So erhalten wir also zwei Kurven, die wir zusätzlich noch Schrittweise miteinander verbinden.

```
Linien (a,b)

/* a im Bereich von 0.0 und 360.0 und b > a */
float a,b;
{
  fx = 75;
  /* Beliebiger x-Schwingungsfaktor */

  fy = 200;
  /* Beliebiger y-Schwingungsfaktor */

  fx1 = 100;
  /* zweites Schwingungsfaktorpaar */
  fy1 = 130;

  Calculate(a);
  /* Initialisierung der ersten 2 Koordinatenpaare */
  for{ n = a;                         /* Schlaufe: von a */
       n < b;                         /*           bis b */
       n+=1;                 /* Schlaufenzähler erhöhen */
       Calculate(n);
                       /* nächste 2 Koordinatenpaare */

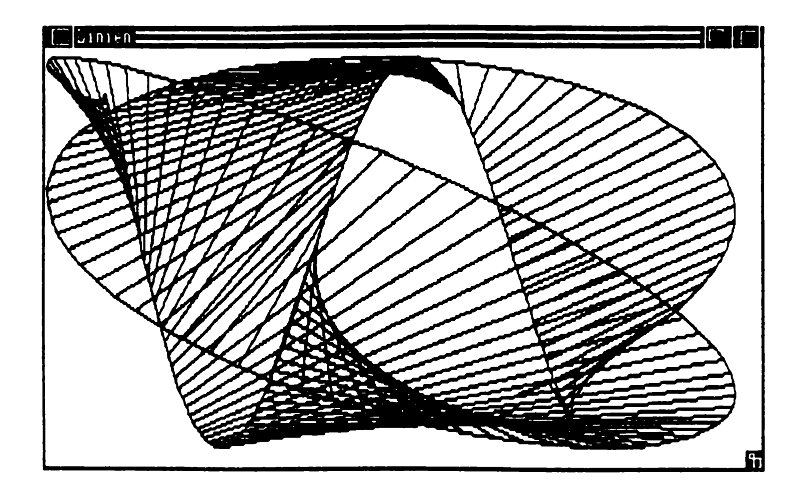
       Line(x,y,x1,y1,farbe);
                            /* Verbinden beider Linien */

       Line(xold,yold,x,y,farbe);          /* 1. Linie */
       Line(xold1,yold1,x1,y1,farbe);      /* 2. Linie */

       xold = x;              /* Zwischenspeichern der */
       yold = y;           /* aktuellen x- und y-Werte */
       xold1 = x1;
                    /* Zwischenspeichern der 2. Werte */
       yold1 = y1;
  };
}
```

```
Calculate (i)
/* Berechnen der aktuellen Koordinaten */
float x,y;
{
  x = sin(i*pi/fx) * 160.0 + 160.0;
                              /* Berechne 1.x-Koordinate */
  y = cos(i*pi/fy) * 115.0 + 115.0;
                              /* Berechne 1.y-Koordinate */
  x1 = sin(i*pi/fx1) * 160.0 + 160.0;
                              /* Berechne 2.x-Koordinate */
  y1 = cos(i*pi/fy1) * 115.0 + 115.0;
                              /* Berechne 2.y-Koordinate */
}
```

Bild 11.2 - Ausgabe des Algorithmus "Linien".



Dieses Bild haben Sie bestimmt schon einmal in dieser oder ähnlicher Form gesehen. Das *Lines*-Demo der Workbenchdiskette funktioniert genauso. Obwohl hier nur Linien gezeichnet werden, entsteht jedoch ein räumlicher Effekt, der dem Bild eine scheinbare Tiefe verleiht. Ähnliche Effekte lassen sich aber auch mit Polygonen oder Kreisen erzeugen. Setzen wir statt der *Line*-Befehle in unserer Schlaufe den *Polygon*-Befehl wie folgt ein, so werden die einzelnen Flächen sofort ausgemalt auf dem Bildschirm erscheinen.

```
Polygon(xold,yold,xold1,yold1,x1,y1,x,y,farbe,TRUE);
```

Die vollständigen Programme zu diesen Algorithmen finden Sie wieder auf der Begleitdiskette, und zwar unter den Namen Linien und Flächen. Abschließend noch ein Beispiel mit Kreisen, wobei nicht nur die Position der einzelnen Kreise von unserer Funktion berechnet wird, sondern auch dessen Radius.

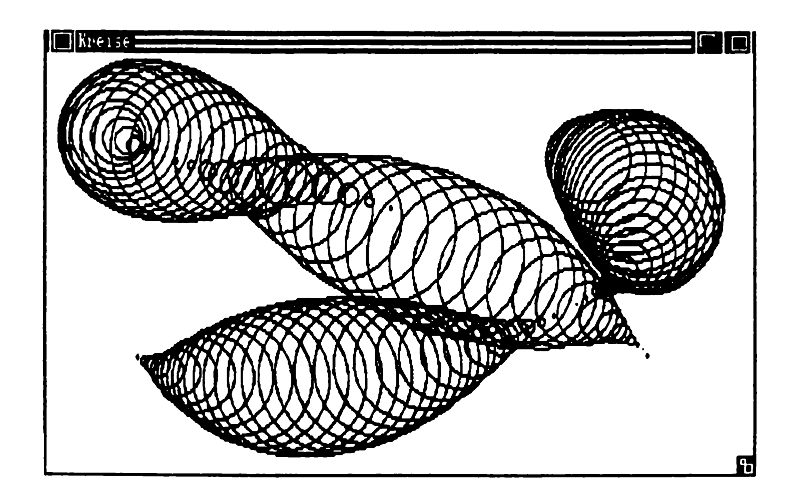
```
Kreise (a,b)
float a,b;
/* a im Bereich von 0.0 und 360.0 und b > a */

{
  fx = 175;              /* Beliebiger
                            x-Schwingungsfaktor */
  fy = 80;               /* Beliebiger
                            y-Schwingungsfaktor */
  fr = 95;               /* Schwingungsfaktor
                            des Radius` */
  for( n = a;            /* Schlaufe: von a */
       n < b;            /*           bis b */
       n++)
     {Calculate(n);
                        /* nächste 2 Koordinatenpaare */
       Circle(x,y,r,farbe);       /* Zeichne Kreis */
```

```
        n+=1;                /* Schlaufenzähler erhöhen */
  };
}
Calculate (i)      /* Berechnen der aktuellen Werte */

{
  x = sin(i*pi/fx) * 110.0 + 160.0;
                             /* Berechne 1.x-Koordinate */
  y = cos(i*pi/fy) * 65.0 + 65.0;
                             /* Berechne 1.y-Koordinate */
  r = sin(i*pi/fr) * 24.0 + 26.0;
                             /* Berechne den Radius */
}
```

Bild 11.3 - Ausgabe des Algorithmus "Kreise"



Auf der Begleitdiskette ist dieser Algorithmus als vollständiges Programm unter den Namen Kreise gespeichert.

Tips:

- Zur Farbgebung dieser Grafiken noch folgendes: anstatt die Farbpalette beliebig zu wählen (wie in den Beispielprogrammen der Begleitdiskette) können Sie den 3D-Effekt dieser Bilder erhöhen, indem Sie nur eine oder zwei verschiedene Farben auswählen und die restlichen Farbregister mit verschieden Intensitäten dieser Farbe(n) belegen.
- Bei der Auswahl der Schlaufenlänge, sowie deren Start- und Endwerte, kann der Computer behilflich sein: benutzen Sie den Zufallsgenerator!
- Für Start- und Endwerte können Sie bestimmte Bedingungen einführen. Im Beispielprogramm "Kreise" haben wir das Schleifenende so gewählt, daß der Radius der Kreise fast Null ist. Somit entsteht immer höchstens ein offenes "Schlauchende".

2. Erzeugen eines Spiralnebels

Einen Sternenhimmel kann man bereits durch zufälliges setzen von mehr oder weniger hellen, weißen Punkten auf einen schwarzen Hintergrund erzeugen. Nun ja, das wird niemanden vom Hocker reißen, der, auf welche Weise auch immer, schon einmal mit einem Weltraumballerspiel konfrontiert worden ist. Wesentlich interessanter wird das Bild, wenn man einen Spiralnebel einsetzt. Zur Verwirklichung benötigt man lediglich eine Funktion, die eine Spirale zeichnet. Diese darf natürlich nicht direkt auf den Bildschirm gebracht

werden, sondern muß "verschmiert" werden. Dazu benutzen wir ein Unterprogramm, daß mit Hilfe von Zufallszahlen eine Sternenwolke erzeugt. Dabei wird die Größe der Wolke durch einen angegebenen Radius bestimmt. Vor dem Setzen eines Punktes überprüfen wir zunächst, ob dieser schon gesetzt ist. Ist dies der Fall so wird sein Farbwert um eins erhöht, so daß er heller erscheint. Die Spiralfunktion schließlich leiten wir aus der Kreisfunktion ab. Ein Kreis entsteht beim Durchlaufen der Funktion x = sin(alpha) und y = cos (alpha) für alpha = 0 bis alpha = 2 * Pi. Eine Spirale erhält man, wenn man jeweils x und y mit einem linear ansteigenden Wert multipliziert. Da sich die Spirale so zu langsam ausdehnt multiplizieren wir jedoch nicht direkt mit der Schlaufenvariablen, sondern mit deren Quadrat. Als vollständiges Programm sieht das dann so aus:

```
/*  SpiralNebel                    von Olaf Pfeiffer,
                              C-Version von Paul Lukowicz  */

#include"Display.h"
#include"gfxTools.h"
#include"intuition/intuition.h"
#include"math.h"

struct Window    *Window;
struct Screen    *Screen;
struct RastPort  *Rast;

int Col[]={ 0, 0, 0,  0, 1, 4,  0, 1, 6,  0, 2, 7,
/* Farbwerte mit unter- */
            0, 2, 9,  1, 2,10,  1, 3,11,  1, 4,12,
/* schiedlichen Inten-  */
            2, 4,12,  2, 5,13,  2, 6,13,  3, 8,14,
/* sitäten für die Dar- */
            4,10,14,  7,11,15, 10,13,15, 15,15,15   /
* stellung der Sterne. */
```

```
          };
main ()
{
 USHORT  code, i, j, h;
 ULONG   Class;

 OpenIntui();
 OpenGfx();

 /*  Einen low-res Screen öffnen.  */
 Screen = (struct Screen *)MakeScr(0,0,320,250,
          "Nebel",4,NULL,NULL,NULL);
 if (Screen == NULL)
  exit(FALSE);

 /*  Ein Fenster auf dem neuen Screen öffnen  */
 Window = (struct Window *)MakeWindow(20,50,270,
          100,0,0,"Spiralnebel",WINDOWCLOSE|ACTIVATE,
          CLOSEWINDOW,Screen);

 if(Window == NULL)    /*  Fehler beim Öffnen ?  */
  exit(FALSE);

 /*  Adresse des Viewports und des Rastports holen */
 Rast = Window->RPort;

 /*  Farbregister initialisieren      */
 SetColors(Screen,&Col,15);

 Background ();         /*  Hintergrund zeichnen  */
 Cloud(160,120,25.0);   /*  Sternwolke Zeichnen   */
 Spirale ();            /*  Spirale Zeichnen      */

 /*  Auf Close-Gadget warten und alles schließen  */
 Class = WaitEvent(Window,&code);
 CloseWindow(Window);
 CloseScreen(Screen);
}
```

```
Background ()  /* Setzt in dem Fenster verschiedene
                  "Sterne" mit  unterschiedlichen
                  Intensitäten  zur  Gestaltung des
                  Hintergrundes.                  */

{
  int n, x, y, c, j;

  j = Random();
  for( n = 1; n <= ((int) (Random() * 150.0) + 200);
  {                                              n++)
    x = (int) ( Random() * 320.0);
/* Zufällige x-Koordinate */
    y = (int) ( Random() * 230.0) + 10;
/* Zufällige y-Koordinate */
    c = (int) ( Random() * 15.0) + 1;
/* Zufällige Intensit*t   */
    SetPixel(Rast,x,y,c);
/* Punkt (=Stern) setzen  */
  };
}


Cloud (x, y, z)  /* Setzt eine Sternenwolke mit
einen von z abhängigem */
                 /* Durchmesser an die Stelle
                       (x,y)                    */

int   x, y;
float  z;
{
  int   n, c, x1, y1;
  float r;

  r = z * (Random()+1.5);
 /* Radius der Sternwolke */
  for( n = 0; n <= ((int) (z * z * (Random() +
1.0)) + 10); n++)
```

```
  {
    x1 = x + (int) (cos(2.0 * pi * Random()) *
r * Random());
    y1 = y + (int) (sin(2.0 * pi * Random()) *
r * Random());
    c = ReadPixel(Window->RastPort,x1,y1);
    if(c < 15)          /* Wird ein Punkt nochmal an
                           die gleiche Stelle */
     ++c;               /* gesetzt, so erhöhe seine
                           Intensität.        */
    SetPixel(Rast,x1,y1,c);
/* Setzte tern */
  };
}


Spirale ()
/* Zeichnet auf einer Spiral-Kurve Sternenwolken. */

{
  float z, hz;

  z = 0.05;
  do {
   hz = 1.0 + z * z * 0.60; /* Schrittweite zwischen
                               den einz. Wolken. */
   Cloud((int) (cos(z)*hz*1.25)+160,(int) (sin(z+pi)
                                       *hz)+120,z);
   Cloud((int) (cos(z+pi)*hz*1.25)+160,(int) (sin(z)
                                       *hz)+120,z);
   z = z + Random() * 0.20 + 0.05;      /* Radius der
                                Wolke vergrößern. */
  } while (z <= 4.25 * pi);
}
```

Die beste Wirkung erzielt das so erstellte Bild, wenn Sie Ihren Monitor etwas dunkler und den Kontrast stärker stellen.

Tips:

- Im vorliegenden Programm sieht man den Spiralnebel direkt von "oben", daß heißt er ist im wesentlichen kreisrund. Von der Erde aus gesehen, sind dies jedoch die wenigsten. Versuchen Sie doch mal durch geeignete Multiplikation der x- und y-Achsen mit einem variablen Wert, einen Spiralnebel zu erzeugen, der die äußere Form einer Ellipse hat, die am besten auch noch diagonal auf dem Bildschirm liegt.

3. Fraktale Kurven und L-Systeme

Neulich stellte uns jemand die Frage: "Was heißt das denn? - Fraktal?". Da dies ein Begriff ist der heute im Zusammenhang mit der Computergrafik zwar oft verwendet aber nur selten erklärt wird, wollen wir dies hier kurz tun: allgemein bezeichnet man damit Gebilde und Erscheinungen (die wir durchaus auch in der Natur wiederfinden) die eine wichtige Gemeinsamkeit besitzen: die Selbstähnlichkeit. Das heißt nichts anderes,als daß man beim vergrößern eines Ausschnittes aus diesem Gebilde wiederum ein dem ursprünglichen ähnliches Gebilde erhält. Bei den fraktalen Kurven geht es in erster Linie um Grafiken, die nur durch eine bestimmte Anordunug von gleichlangen Strichen entstehen, die alle miteinander in bestimmten Winkeln verbunden sind. Dabei gibt es praktisch nur drei Zeichenbefehle, mit denen der Grafik-Cursor gesteuert wird. Diese Ansteuerung erfolgt durch Interpretation einer Zeichenkette, wobei alle Zeichen der Kette durchgegangen werden. Ist das folgende Zeichen ein "F", so bewegt sich der Cursor um eine Längeneinheit nach vorne (forward) und zeichnet so eine Linie. Ist das nächste Zeichen ein "+" oder ein "-" so dreht sich der Cursor um eine Winkeleinheit delta nach rechts oder links. Andere Zeichen sollen zunächst einmal ohne Wirkung bleiben. Das Wesentliche bei dieser Technik ist aber die Enttehung dieser Zeichenkette. Dabei wird die Zeichenkette zunächst einmal Initialisiert, ihr werden also ein oder mehrere Zeichen zugeordnet. Diese Zeichenkette werden wir ab jetzt Axiom nennen. Als nächstes gibt es eine oder mehrere Regeln, die jeweils ein bestimmtes Zeichen des Axioms durch eine andere Zeichenkette ersetzten. Diese Regeln werden in einer

Schlaufe auf das ganze Axiom angewendet.Dazu erstmal ein Beispiel: Axiom sei "F++F++F", daß heißt bei einer Winkeleinheit von Delta = pi/3 (60 Grad) erhalten wir bei der oben definierten Interpretation ein gleichseitiges Dreieck.

Lautet die Regel *"F"->"F-F++F-F"*, so bedeutet das, daß jedes "F" aus unserem Axiom durch die Zeichenkette *"F-F++F-F"* ersetzt werden soll. Lassen wir diese Regel auf unsere Zeichenkette los, so erhalten wir als neue Zeichenkette:

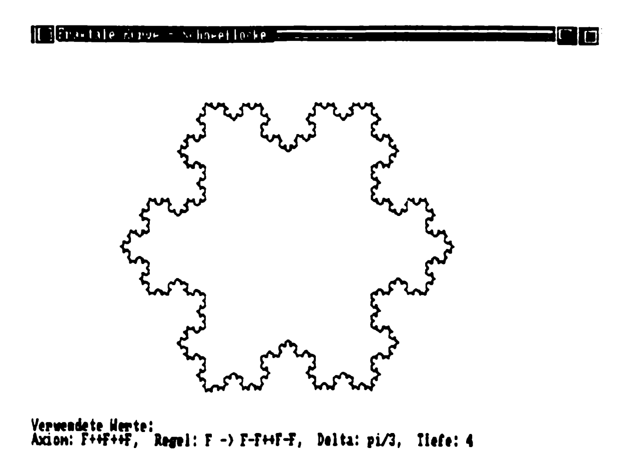
"F-F++F-F++F-F++F-F++F-F++F-F"

Schicken wir unseren Grafik-Cursor los, dann erhalten wir einen sechszackigen Stern. Nun kann man aber diese Regel beliebig oft anwenden und erhält somit nicht nur wesentlich längerer Zeichenketten, sondern auch dementsprechend "verwinkeltere" Figuren. In unserem Fall sähe die Zeichenkette nach dem nächsten Schritt so aus:

"F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++
F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++
F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F"

Lassen wir die Regel noch ein paarmal auf unsere Zeichenkette los, so entsteht eine Art Schneeflocke, wie in Bild 11.4.

Bild 11.4 - Die Schneeflocken-Kurve.



In den Bildern 11.5, 11.6 und 11.7 finden sie weitere Beispiele mit anderen Axiomen und Reproduktionsregeln. Die jeweiligen verwendeten Werte finden Sie ebenfalls in den Bildern abgedruckt.

Bild 11.5 - Noch eine Schneeflocke, diesmal die Spitzen nach innen "geklappt".

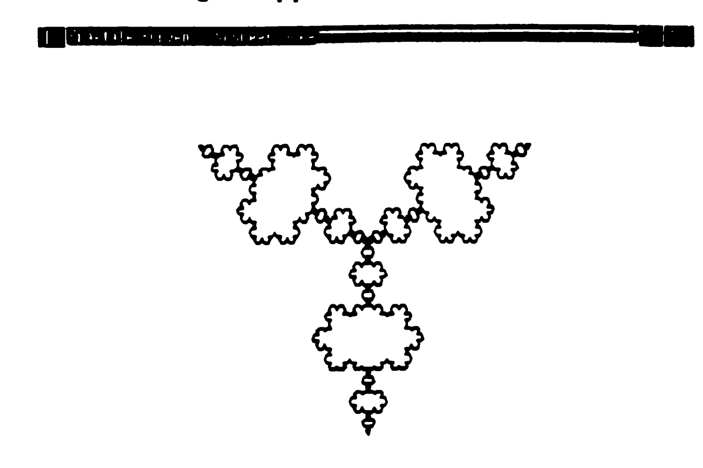
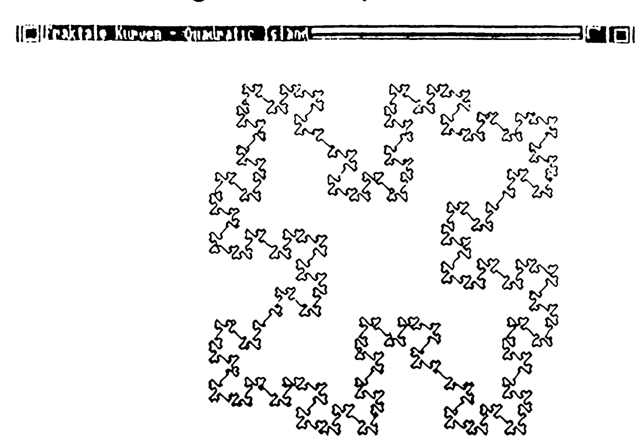
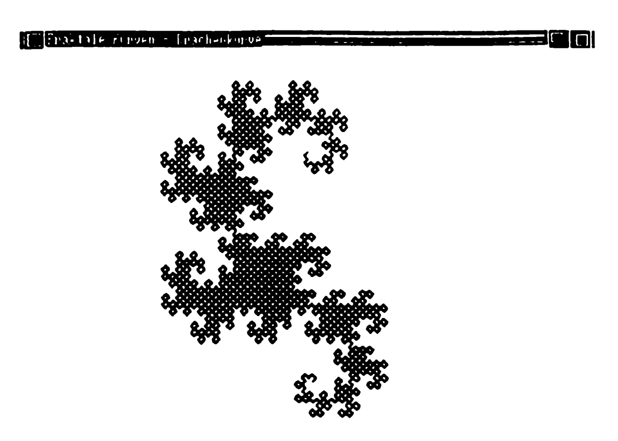


Bild 11.6 - Das sogenannte "Quadratic Island".



Axiom: F+F+F+F, Regel: F -> F+F-F-FFF+F+F-F, Delta: pi/2, Tiefe: 3

Bild 11.7 - Die sogenannte "Drachenkurve" benötigt bereits zwei verschiedene Reproduktionsregeln.

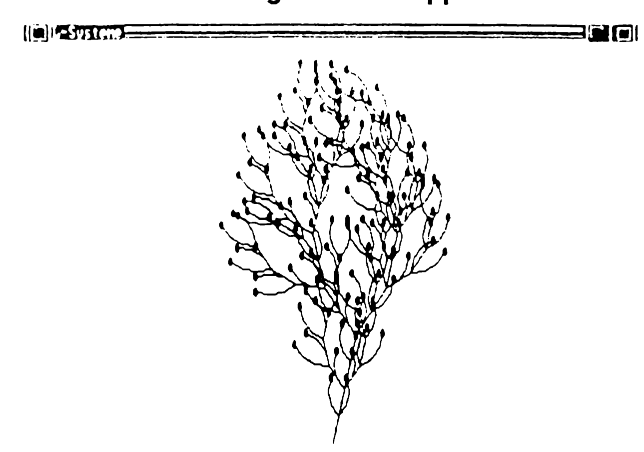


Axiom: X, Regeln: X -> X+YF+, Y -> -FX-Y, Delta: pi/2, Tiefe: 13

Bei den L-Systemen (benannt nach Lindenmayer) benutzt man die gleiche Technik wie bei den Fraktalen Kurven, es werden lediglich noch zwei weitere Zeichen und deren Interpretationen eingeführt.

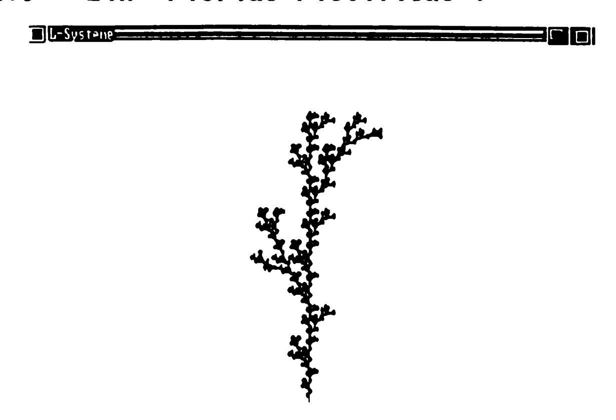
Das Zeichen "[" soll heißen: hier beginnt eine Verzweigung und das Zeichen "]": hier endet ein "Ableger", deshalb gehe zurück zur letzten Verzweigung und mache dort weiter. Somit ist es möglich, fiktive Pflanzen zu erzeugen, die teilweise eine verblüffende Ähnlichkeit zu existierenden Pflanzen haben (Siehe Bild 11.8 und 11.9). Programmtechnisch muß also ein Stack (Zwischenspeicher, der die zuletzt eingegebenen Werte zuerst wieder ausgibt) angelegt werden, der sich an einer Verzweigung "[" nicht nur die aktuelle Position merkt, sondern auch die Richtung, in die der Cursor sich weiterbewegen würde. Bei einem "]" müssen sie dann wieder als die aktuelle Position und Richtung benutzt werden. In dem hier vorliegenden Programm müssen Sie außerdem noch die Startkoordinaten und die Startrichtung angeben, von der aus mit dem Zeichnen begonnen werden soll.

Bild 11.8 - Ein "buschiges Gestrüpp".



Verwendete Werte:
Axiom: F], Regel: F -> FF+[+F-F-F]-[-F+F+F], Delta: pi/8, Tiefe: 3

Bild 11.9 - Ein "Florius Fictivicus".



Zur grafischen Interpretation der erzeugten Zeichenkette empfielt sich die Verwendung von sogenannten Turtle-Befehlen, mit denen ein fiktiver Zeichenstift über den Bildschirm geschickt werden kann. Der Name Turtle (engl.:Schildkröte) läßt erahnen, wie langsam die ersten Implementationen dieser Grafikbefehle gewesen sein müssen...

Da der Amiga standardmäßig über diese Befehle nicht verfügt, geben wir hier nur einen vereinfachten "Pseudo-Algorithmus" zur grafischen Interpretation der Zeichenkette an. Neben den Turtle-Befehlen werden Funktionen zur String- (**Length**(String) liefert die Länge einer Zeichenkette und **GetChar**(n,String) das nte Zeichen der Zeichenkette) und Stack-Behandlung benutzt (**Push**(value) legt den Wert **value** auf den Stack ab, **Pop** holt den obersten wieder vom Stack runter).

```
InitTurtle
                          /* Den Zeichenstift initialisieren,
                             also an eine bestimmte Position
                             setzen und eine Bewegungsrichtung
                             festlegen. */

for n := 1 to Length(String) do
                         /* In einer Schlaufe alle Zeichen der
                            Zeichenkette durchgehen. */

  char := GetChar(n,String);
                         /* n-tes Zeichen aus der Zeichenkette
                            holen. */

  if    char = "F" then MoveForward
                         /* Bei "F" Zeichenstift vorwärts
                            bewegen. */

  elsif char = "+" then TurnRight
                         /* Bei "+" die Bewegungsrichtung des
                           Zeichenstiftes um eine Einheit nach
                            rechts drehen. */

  elsif char = "-" then TurnLeft
                         /* Bei "-" die Bewegungsrichtung des
                           Zeichenstiftes um eine Einheit nach
                            links drehen. */
  elsif char = "[" then Push(Direction)
                         /* Bei "[" die Bewegungsrichtung des
                            Zeichenstiftes, sowie */

                                 Push(YPosition)
                         /* die y-Koordinate und */
                                 Push(XPosition)
                         /* die x-Koorinate der Position  des
                        Zeichenstiftes auf den Stack legen. */
```

```
    elsif char = "]" then PenUp
                    /* Bei "]" den Zeichenstift aufheben, */

                              MoveTo(Pop,Pop)
                    /* ihn zu der zuletzt auf den Stack
                       gelegten Position bewegen und */

                              TurnTo(Pop)
                    /* ihm die vorherige Bewegungsrichtung
                       zurückgeben. */

                              PenDown
                    /* Abschließend Zeichenstift wieder
                       absetzen. */
end
```

Tips:

- Auch hier spielt die Farbwahl wieder eine große Rolle. Bei den L-Systemen erscheint es sinnvoll die Farbe von der Verzweigungstiefe abhängig zu machen. So kann man die Äste immer heller, bzw. grüner machen als den Stamm.

- Die Enden der Zweige sollten sie mit Blättern oder gemischt mit Blüten verzieren. Dazu müssen sie lediglich eine Unteroutine schreiben, die z.B. an der aktuellen Position einen bunten Kreis setzt und jedesmal bei dem Zeichen "]" mitaufgerufen wird.

4. Random Pixels - wielange darfs denn dauern ?

Lassen Sie uns hier zunächst bemerken, daß es grundsätzlich zwei Arten von Grafikprogrammierern gibt: die einen Schalten Ihr "komplexes Rechenwerk" an, geben ein paar komplexe Werte in Ihr komplexes Programm und - gehen ins Bett... Am nächsten Morgen (manchmal auch erst am übernächsten) ist es dann soweit: "Oh was für eine schöne, komplexe Darstellung einer komplexen Zahlenmenge!" - Und dann sind da noch die Grafikprogrammierer, die ohne "komplexe" arbeiten: diese muten ihrem Computer nur soviel zu, wie er in wenigen Minuten anhand schneller Algorithmen erledigen kann. - Bevor da irgendein Zweifel aufkommt: wir gehören zu den letzteren!

Wie sie Anhand dieser Einleitung sicher schon bemerkt haben, kommen wir zu einer Technik die je nach Anwendung sehr lange dauern kann. Der Grundgedanke dabei ist folgender: An einer mehr oder weniger zufälligen Stelle am Rand des Fensters wird ein Pixel losgeschickt. Losgeschickt deshalb, weil er sich ab nun jeweils zufällig in eine beliebige Richtung mit der Schrittweite eins weiterbewegt. Dieser Vorgang wird solange wiederholt, bis er entweder das Fenster verläßt (dann wird ein neuer losgeschickt) oder er auf einen anderen Pixel stößt. Ist dies der Fall, so friert er an dieser Stelle fest und erhält bestenfalls noch eine bestimmte Farbe. Die Zeit, die das Programm zur Erstellung einer solchen Grafik braucht, wird dabei im wesentlichen durch die Dimensionen des Fensters und der Anzahl der anfangsgesetzten Pixel bestimmt.

Öffnen sie also ein größtmögliches Fenster und setzten zum Beginn nur ein Punkt in der Mitte, so können sie bei diesem Algorithmus, wenn Sie Pech haben, erst nach einer halben Stunde das atemberaubende Ereignis miterleben, wie zwei Pixel miteinander verschmelzen! So wird dieser Algorithmus auch in einer deutschsprachigen Wissenschaftszeitschrift auf den Leser losgelassen. Immerhin wird in diesem Artikel auch zugegeben, daß dieser Algorithmus auf einem IBM XT "vier bis fünf Stunden" braucht. Dabei kann man durch ein paar Veränderungen den Algorithmus um einiges schneller gestalten, ohne die entstehende Grafik grundlegend zu verändern. Anstatt einen Pixel am Rande des Fensters zu Starten, schicken wir ihn vom äußeren Rand der entstehenden Figur los. Darüberhinaus lassen wir den Pixel nicht bis zum Fensterrand wandern, ehe wir ihn für ungültig erklären, sondern betrachten ihn bereits als verloren, wenn er um einen gewissen Betrag hinter den eben erwähnten äußeren Rand gewandert ist. Wählt man die Dimension nicht allzu groß, erhält man bereits nach wenigen Minuten die fertige Grafik, eine Art Korallenstock. Das so verschnellerte vollständige Programm sieht wie folgt aus:

```
/* RandomPixels

    von Olaf Pfeiffer, C-Version von Paul Lukowicz */

#include"Display.h"
#include"gfxTools.h"
#include"intuition/intuition.h"
#include"math.h"

#define Modus 2
struct Window    *Window;
struct Screen    *Screen;
```

```
struct RastPort  *Rast;

/* Setzen der Farben */
int Col [] = { 0, 0, 0,  1,13, 3,  0, 4, 4,  0, 4, 3,
               1, 5, 2,  2, 5, 1,  2, 6, 0,  3, 6, 1,
               4, 7, 1,  4, 7, 2,  5, 8, 2,  6, 8, 3,
               6, 9, 4,  7, 9, 5,  7,10, 4,  8,10, 5,
               8,11, 6,  9,11, 7,  9,12, 6, 10,12, 7,
              10,13, 8, 11,13, 9, 11,14, 8, 12,14, 9,
              12,15,10, 13,15,11, 13,14,10, 14,13,11,
              14,12,12, 15,11,13, 15,10,14, 15, 9,15
             };

int Dim, Dimx, Dimy,
  /* Dimensionen der Grafik */

    rfig, rmax,
  /* aktueller und maximaler Radius der Figur */

    x, y, xalt, yalt;
  /* aktuelle und alte Koordinaten des Punktes */

main ()
{
 USHORT  code, i, j, h;
 ULONG   Class;
 int     xalt, yalt, Cont, Color;
 char    ch;

 OpenIntui();
 OpenGfx();

 /*  Einen low-res Screen öffnen.  */
 Screen = (struct Screen *)MakeScr(0,0,320,250,
             "RandomPixels",5,NULL,NULL,NULL);
 if(Screen == NULL)
  exit(FALSE);
```

```
/*  Ein Fenster auf dem neuen Screen öffnen  */
Window = (struct Window *)MakeWindow(20,50,270,100,
          0,0,"RandomPixels",WINDOWCLOSE|ACTIVATE,
          CLOSEWINDOW,Screen);

if(Window == NULL)    /*  Fehler beim Öffnen ?  */
 exit(FALSE);

/*  Adresse des Rastports holen  */
Rast = Window->RPort;

/*  Farbregister initialisieren      */
SetColors(Screen,&Col,15);

Dimx = Dim / 2 + 5;
            /* Berechnen der halben x-Dimension */

Dimy = Dimx + 10;
            /* Berechnen der halben y-Dimension */

Line(Rast,Dimx-1,Dimy-1,Dimx+1,Dimy+1,2);
            /* initialisieren der Figur */
rfig = 1;
            /* aktueller Radius der Figur */
rmax = 81;
            /* maximaler Radius der Figur */
do
{
  StartPixel ();
           /* Schicke einen Punkt auf die "Reise" */
  do
  {
    RandomMove ();           /* Bewege den Punkt */
    Cont = Valid(x,y,rmax);
                      /* überprüfe die Gültigkeit */

   Color = Found(x,y) + 1;
            /* Durchschnittsfarbwert der Umgebung */
```

```
    } while( (Cont == TRUE) && (Color <= 1));
                        /* bis Ungültig oder Kollision */

   if( Color > 32 )
                        /* größter Farbwert erreicht ? */
     Color = 2;         /*  also Farbwert zurücksetzen */
    if(Cont = TRUE)
                /* Es gab eine Kollision mit der Figur */
    {
      SetPixel(Rast,x,y,Color);
                   /* Punkt mit der neuen Farbe setzen */

      if(Valid(x,y,(rfig * rfig)) == FALSE)

      {           /* Überprüfen der Radien der Figur */
        rmax = rmax + 2 * (rfig + 8) + 1;
                  /* berechnen des neuen max. Radius */
        ++rfig;
                           /* aktuellen Radius erhöhen */
      };
    }
    else
                              /* Der Punkt hat den     */
                              /* Gültigkeitsbereich    */
    SetPixel(Rast,x,y,0);    /* verlassen, also wieder */
                              /* löschen.              */
  } while( rfig > Dim / 2 - 10);
              /* wiederholen bis Radius > Dimension */

/*  Auf Close-Gadget warten und alles schließen  */
 Class = WaitEvent(Window,&code);
 CloseWindow(Window);
 CloseScreen(Screen);

}
```

```
/* Liegt der Punkt (x,y) im Gültigkeitsbereich ? */

int Valid(x, y, rq)

int x, y, rq;
          /* x- und y-Position, rfig zum Quadrat */

{
  int ok;                    /* Rückgabe-Variable */

  x = x - Dimx;      /* x-Entfernung vom Ursprung */
  y = y - Dimy;      /* y-Entfernung vom Ursprung */
  if( x * x + y * y < rq)
   ok = TRUE;        /* Punkt ist gültig */
    else
   ok = FALSE;       /* Punkt ist ungültig */
  return ok;
}

/* Prüfen auf Kollision */

int  Found(x, y)

int x, y;     /* x- und y-Koordinaten des Punktes */

{
  int a, b,
          /* Schleifenzähler für die Nachbarpunkte */

      z, zg,
          /* Zähler für gesetzte Nachbarpunkte und
             deren Farbwerte */

      c;  /* aktueller Farbwert eines
             Nachbarpunktes */

  z = 0;               /* Initialisieren der Zähler */
  zg = 0;
```

```
  for( a = -1; a <= 1; a++)
  {                                   /* linke und rechte
                                         Nachbarpunkte */
    for( b = -1; b <= 1; b++)
    {                                 /* obere und untere
                                         Nachbarpunkte */
     if( (a != 0) && (b != 0) )
      {                               /* den Punkt selbst
                                         ausschließen */
        c = ReadPixel(Rast,x+a,y+b);
        if( c > 0 )
        {                             /* ein Nachbarpunkt
                                         ist gesetzt */
          zg += c;                    /* Farbwerte
                                         aufaddieren */
          ++z;                        /* Zähler
                                         incrementieren */
        };
      };
    };
  };
  if( z != 0 )                    /* Falls Punkt gesetzt, */
   zg = zg / z;                   /* Durchschnittsfarbwert
                                     berechnen */
  return zg;                      /* und zurückgeben */
}

/* Einen Punkt auf die "Reise" schicken */

StartPixel ()

{
  float alpha, rf;          /* Startwinkel und -radius */

  do
  { alpha = Random() * 2.0 * pi;
                                /* zufälliger Startwinkel */
```

```
    rf = (float) (rfig) + 5.0;
                        /* und zufälliger Startradius */

    x = (int) (sin(alpha) * rf -
                    Random() * rf / Modus) + Dimx;

    y = (int) (cos(alpha) * rf -
                    Random() * rf / Modus) + Dimy;
  } while( Found(x,y) != 0);
                        /* wiederholen bis Punkt gültig */

  SetPixel(Rast,x,y,1);              /* Punkt setzen */
   xalt = x;              /* und Koordinaten merken */
   yalt = y;
}

/* Bewegen des Punktes um einen Schritt */

RandomMove ()
{
  float b;

  b = Random();
  if(b < 0.25 )
   ++y;                        /* y inkrementieren */
  else if( b < 0.5  )
   --(y);                      /* y dekrementieren */
  else if( b < 0.75 )
   ++x;                        /* x inkrementieren */
  else
   --(x);                      /* x dekrementieren */
  SetPixel(Rast,xalt,yalt,0); /* alten Punkt
                                 löschen */
  SetPixel(Rast,x,y,1);       /* und an neuer
                                 Position setzen */
  xalt = x;                   /* Koordinaten merken */
  yalt = y;
}
```

Bild 11.10 - Ein durch den Algorithmus "Randomixels" erzeugter "Korallenstock".





Für die Dimension 75 rechnet der Amiga (ohne Abschalten anderer Tasks) unter drei Minuten, allerding benötigt er für die Dimension 200 immernoch etwas über eine halbe Stunde. Deshalb haben wir in unserem vollständigen Programm **RandomPixels** der Begleitdiskette einen weiteren Verschnellerungsfaktor eingebaut, der jedoch die entstehende Figur in einem stärkeren Maße beeinträchtigt. Ein Pixel wird dann nicht mehr nur auf den "Umkreis" der Figur losgeschickt, sondern innerhalb eines Ringes gestartet, der in die Figur hineinreicht. Dabei ist der äußere Radius dieses Ringes der gleiche wie vorher und der innere um den Flächenfüllfaktor kleiner. Diesen Namen haben wir ihm gegeben, da die so entstehenden Figuren sich bei gleichem Umfang aus wesentlich mehr Pixel zusammensetzen.

Tips:

- Zur Initialisierung kann man auch mehrere Pixels oder gar Linien verwenden, die nicht unbedingt in der Mitte des Bildschirms stehen müssen. Beachten Sie dann aber, daß Sie die Definition des äußeren Randes der Figur eventuell neu vornehmen müssen. So erhalten Sie zum Beispiel durch setzen einer Linie am unteren Fensterrand und entsprechender Definition der Start- und Abbruchbedingungen Gewächse, die an Unterwasserpflanzen erinnern.

- Ein Vorschlag der nochmals eine wesentliche Verschnellerung bewirkt: sorgen Sie dafür, daß ein Pixel sich immer auf die Figur zubewegt und sich nicht von ihr entfernt. Zugegeben, bei unserer ursprünglichen Figur ist das eine nicht ganz so einfache Aufgabe, aber um so leichter ist es bei den eben angesprochenen Gewächsen. Erhöhen Sie in der Prozedur **RandomMove** den Vergleichswert 0.25 vor dem INC(y) von auf einen Wert kleiner als 0.5. Je näher dieser Wert bei 0.5 liegt, desto größer die Wahrscheinlichkeit, daß sich der Pixel nach unten bewegt.

- Das wesentliche Erscheinungsbild können Sie auch auf folgende Art verändern: fügen Sie bei der Kollision einen Zufallswert ein, ob der Punkt hier einfrieren soll oder nicht. Dadurch werden die einzelnen Äste der Figur dicker, wobei wir jedoch nicht vergessen dürfen zu erwähnen, daß dadurch das Programm wieder langsamer wird.

Kapitel 12

==
Techniken der dreidimensionalen Grafikerzeugung
==

Während wir uns im letzten Kapitel nur mit der zweidimensionalen Grafik beschäftigt haben, werden wir hier anhand einiger Beispiele zeigen, wie man dreidimensionale Objekte auf dem Bildschirm darstellt. Am Ende der Abschnitte mit den Programmbeispielen finden sie wieder Tips, die Ihnen Anregungen zur Veränderung der Programme geben.

1. Das Koordinatensystem

Schlägt man in einem Grafikbuch das erste Kapitel über dreidimensionale Grafik auf, so wird man meißtens mit jeder Menge Mathematik konfrontiert. Dort stehen dann oft seitenlange Einführungen in die Vektorrechnung die manchmal sogar über den dreidimensionalen Raum hinausgeht. Da eine Darstellung von mehr als drei Dimensionen für uns hier absolut unbedeutend ist, betrachten wir also ab jetzt ausschließlich den dreidimensionalen Raum. Wir verzichten an dieser Stelle bewußt auf die Theorie der Vektorrechnung und gehen davon aus, daß Ihnen ein "Vektor in einem dreidimensionalen Raum" ein bekannter Begriff ist. Auch sollten Sie sich darüber klar sein, daß jeder Körper im Raum durch Vektoren dargestellt werden kann. Ansonsten werden wir uns auf die reine Anwendung der entsprechenden Formeln beschränken, ohne Sie erst zu beweisen, da Sie für uns reine Hilfsmittel seien sollen. Zunächst aber erst einmal zu den beiden Definitionen der drei Koordinatenachsen. Bei der mathematischen Definition geht man von eine (x,y)-Ebene aus, auf der die z-Werte senkrecht aufgetragen werden. Auf Computern trifft man jedoch oft auf folgende Definiton der x-, y- und z-Achse:

Als x- und y-Achse definieren wir die gleichen wie im zweidimensionalen. Sitzt also der Ursprung dieses Koordinatensystems in der Bildschirmmitte, so geht die x-Achse nach rechts und die y-Achse nach oben! Die z-Achse ist nun die dritte, neue Achse, die jedem Punkt auf dem Bildschirm auch noch eine "Tiefe" zuordnet. Diese verläuft also in Ihren Monitor "hinein". Bei diesen beiden Systemen werden also die y- und z-Achsen miteinader vertauscht. Abgesehen von der Berechnung mathematischer Funktionen werden wir uns im wesentlichen an das zweite System halten.

2. Die Rotationsmatrix

Hierbei handelt es sich um ein Hilfsmittel, daß es uns erlaubt einen Vektor und somit einen beliebigen Punkt im Raum um einen beliebigen Winkel um die drei Koordinatenachsen zu drehen. Damit ist es dann auch möglich, beliebige Körper und Flächen auf dem Bildschirm zu drehen. Da in vielen Anwendungen nur eine Drehung um ein oder zwei Winkel nötig ist, kann diese Matrix verkürzt werden, indem die nichtveränderbaren Winkel direkt eingesetzt und ausgerechnet werden. Dies haben wir auch in den nächsten beiden Abschnitten so gehandhabt.

Hier nun die Definition der Rotationsmatrix:

```
RM[1,1]= cos(wz)* cos(wy);
RM[2,1]=-cos(wz)* sin(wy);
RM[3,1]= sin(wz);
```

```
RM[1,2]= sin(wx)*sin(wy)-cos(wx)*cos(wy)*sin(wz);
RM[2,2]= sin(wx)*cos(wy)+cos(wx)*sin(wy)*sin(wz);
RM[3,2]= cos(wx)*cos(wz);
RM[1,3]= cos(wx)*sin(wy)+sin(wx)*cos(wy)*sin(wz);
RM[2,3]= cos(wx)*cos(wy)-sin(wx)*sin(wy)*sin(wz);
RM[3,3]=-sin(wx)*cos(wz);
```

Dabei sind ***wx***, ***wy*** und ***wz*** die gewünschten Drehwinkel um die jeweilige Achse (im Bogenmaß). Beachten Sie dabei, daß die Drehungen um die x-Achse nach vorne, um die y-Achse nach rechts und um die z-Achse im Uhrzeigersinn erfolgen! Die Übertragung auf unser definiertes Koordinatensystem erhalten wir durch folgende Multiplikation mit den jeweiligen Koordinaten x, y und z:

```
xnew = RM[1,1] * x + RM[2,1] * y + RM[3,1] * z;
ynew = RM[1,2] * x + RM[2,2] * y + RM[3,2] * z;
znew = RM[1,3] * x + RM[2,3] * y + RM[3,3] * z;
```

Dabei ist es selten erforderlich ***znew*** wirklich auszurechnen, da uns zur Darstellung auf dem Bildschirm meißtens die x- und y-Koordinate eines Punktes ausreicht.

Als Anwendungsbeispiel nehmen wir den Punkt (10,20,0). Gehen wir von unserem Ursprung in der Bildschirmmitte aus, so liegt dieser Punkt 10 nach rechts, 20 nach oben und 0 nach hinten vom Ursprung. Gedreht werden soll er: 30 Grad nach vorne um die x-Achse, 90 Grad nach rechts um die y-Achse und 45 Grad im Uhrzeigersinn um die z-Achse. Die neuen Koordinaten ***xnew***, ***ynew*** und ***znew*** erhalten wir, wenn wir die Winkel ***wx*** := Pi/3, ***wy*** := Pi, ***wz*** := Pi/2 setzen und anschließend die oben gezeigte Multiplika-

tion mit den Koordinatenpunkten durchführen. So erhalten wir den "gedrehten" Punkt: (10,5,13).

3. 3D - Darstellung von Objekten

Wir werden hier als Beispiel das Koordinatensystem auf dem Bildschirm darstellen und es um beliebige Winkel drehen. Dazu benötigen wir zunächst die x-, y- und z-Koordinaten der Punkte die durch Linien oder Linienzüge miteinander verbunden sind. Haben wir alle Punkte erfaßt, so lassen wir lediglich noch die Rotationsmatrix mit den gewünschten Winkeln auf alle diese Punkte los und erhalten die "gedrehten" Werte. Den Wert ***znew*** berechnen wir dabei nicht, da wir nur die x- und y-Koordinaten zur Bildschirmdarstellung benötigen.

```
/*     Koordinaten          */

#include"Display.h"
#include"gfxTools.h"
#include"intuition/intuition.h"
#include"math.h"

struct Window     *Window;
struct RastPort   *Rast;
float wx,wy,wz;          /* Die Drehwinkel um die einzel-
                              nen Koordinatenachsen */
int     Posx,Posy,Posx1,Posy1,xm,ym,n;
char     ch;
```

```
float L [18] [3] ={0,0,0, 100,0,0,  90,0,0, 100,0,10,
100,0,0,  90,0,10,
                   0,0,0, 0,100,0,  0,90,0, 10,100,0,
10,100,0, 10,90,0,
                   0,0,0, 0,0,100,  0,0,90, 0,10,100,
0,5,95,   0,10,90
                   };
/* Die Vektorenpaare (x1,y1,z1,x2,y2,z2) die jeweils
                    eine Linie im Raum definieren. */
float M [3] [3];     /* Die Rotationsmatrix */


main ()
{
 USHORT  code, i, j, h;
 ULONG   Class;

 OpenIntui();
 OpenGfx();

  printf(" \n >> Erstellung eines 3D-Koordinatensys-
                                   tems          << \n");
  printf("======================================== \n");
  printf(" Drehwinkel um die x-Achse: ");
  scanf("%d ",&n);
  wx = n * pi / 180.0;
  printf(" Drehwinkel um die y-Achse: ");
  scanf("%d ",&n);
  wy = n * pi / 180.0;
  printf(" Drehwinkel um die z-Achse: ");
  scanf("%d ",&n);
  wz = n * pi / 180.0;
  printf(" Drehwinkel um die Achsen: %e %e %e \n ",
  wx,wy,wz);

 /*  Ein Fenster auf dem Workbench-Screen öffnen  */
 Window = (struct Window *)MakeWindow(0,0,639,230,0,
                                    0,"Koordinaten",
```

```
WINDOWCLOSE¦ACTIVATE,CLOSEWINDOW,NULL);
 if(Window == NULL)    /*  Fehler beim Öffnen ?  */
exit(FALSE);
 /*  Adresse des Viewports und des Rastports holen */
 Rast = Window->RPort;

 RotMatrix ();           /* Die zugehörige Rotationsma-
trix ausrechnen */

 for( n = 0; n <= 17; n+=2 )
 {
   Transfer(L[n][0],L[n][1],L[n][2]);
/* Startpunkt einer Linie */
   Posx1 = Posx; Posy1 = Posy;
/* Bildschirmkoordinaten merken */
   Transfer(L[n+1][0],L[n+1][1],L[n+1][2]);
/* Endpunkt einer Linie */
   Line(Window->RPort,Posx,Posy,Posx1,Posy1,n / 6+1);
  };

 /*  Auf Close-Gadget warten und alles schließen  */
 Class = WaitEvent(Window,&code);
 CloseWindow(Window);
}

RotMatrix ()

{
   M[0][0] = cos(wz) * cos(wy);
   M[1][0] =-cos(wz) * sin(wy);
   M[2][0] = sin(wz);
   M[0][1] = sin(wx) * sin(wy) - cos(wx) * cos(wy)
                     * sin(wz);
   M[1][1] = sin(wx) * cos(wy) + cos(wx) * sin(wy)
                     * sin(wz);
   M[2][1] = cos(wx) * cos(wz);
   M[0][2] = cos(wx) * sin(wy) + sin(wx) * cos(wy)
                     * sin(wz);
```

```
   M[1][2] = cos(wx) * cos(wy) - sin(wx) * sin(wy)
                     * sin(wz);
   M[2][2] =-sin(wx) * cos(wz);
}
Transfer(x,y,z)
/* Einen Punkt im 3D-System mit Hilfe der
                        Rotationsmatrix drehen. */
float x, y, z;
{
   float x1, y1, z1;
   x1 = M[0][0] * x + M[1][0] * y + M[2][0] * z;
   y1 = M[0][1] * x + M[1][1] * y + M[2][1] * z;
   z1 = M[0][2] * x + M[1][2] * y + M[2][2] * z;
   Posx = 320 + (int) x1 *2;  /* Umrechnung in Bild-
schirmkoordinaten */
   Posy = 130 - (int) y1;
}
```

4. 3D - Funktionen

Zunächst nochmals kurz zur Erinnerung: Ein "normaler" Graf einer Funktion y = f(x) hat zwei Koordinatenachsen, wird also zweidimensional dargestellt. Redet man von einer 3D-Funktion so ist damit eine Funktion der Form z = f(x,y) gemeint.

Anschaulich bilden die Koordinatenachsen x und y eine Ebene, auf der sich die Funktion, z.B. in Form von Sinusschwingungen, erhebt. Programmtechnisch werden wir also einen doppelt dimensionierten Array in Abhängigkeit von x und y anlegen, der zu den Koordinatenpaaren (x,y) den entsprechenden Funktionswert enthält. Das Programm läßt sich in zwei wesentliche Abschnitte teilen: das Ausrechnen der einzelnen Funktionswerte und die Darstellung auf den Bildschirm.

Beachten Sie bei Funktionen bitte immer, daß Sie einen sinnvollen Definitionsbereich auswählen, so daß es gar nicht erst zu einem "Division by Zero"-Error kommen kann.

Die allgemein übliche Form der Ausgabe solcher Funktionen ist die eines Gitternetzes, welches sich aus vielen kleinen Vierecken zusammensetzt. Betrachten wir jedes dieser Vierecke für sich, so haben wir Polygone, die durch ihre Eckkoordinaten gegeben sind. Diese können wir bereits auch beliebig gedreht auf dem Bildschirm darstellen. Um das gesamte Gitternetz auszugeben, müssen wir lediglich eine Schlaufe programmieren, die alle Vierecke zeichnet. Zuvor lohnt sich aber noch eine Überlegung zum Thema verdeckte Linien: Linien, die im Hintergrund liegen, also durch davorstehende verdeckt werden, sollen auch wirklich nicht erscheinen.

Um jetzt aber vor dem Zeichnen einer Linie nicht erst nachrechnen zu müssen, ob diese sichtbar ist oder nicht, bedienen wir uns eines einfachen Tricks. Wir zeichnen nicht einfache Polygone, sondern umrandete, mit der Hintergrundfarbe ausgefüllte Polygone. Nun legen wir die Schlaufen, die die einzelnen Vierecke ausgeben, so, daß die hintenstehenden Vierecke zuerst, die vornestehenden zuletzt gezeichnet werden. Sollte es "verdeckte" Linien geben, dann werden sie bei diesem System von den im Vordergrund stehenden übermalt.

Auch bei der beliebigen Drehung des Gitternetzes sind einige Einschränkungen durchaus sinnvoll. Auf eine Drehung um die von uns definierte z-Achse können wir verzichten, wodurch sich die Rotationsmatrix um einiges vereinfacht, wenn wir für den Winkel ***wz*** sofort 0 einsetzten.

```
/* Funktionen-Plotter */

#include"Display.h"
#include"graphics/gfx.h"
#include"intuition/intuition.h"
#include"math.h"
#include"AreaExtras.h"

#define Dim    64   /* Die Größe des Koordinaten-
                                           netzes */
#define pi     3.14159

struct Coordinates
        {
         SHORT xK,yK;          };


struct Window    *Window;
struct Screen    *Screen;
struct RastPort  *Rast;
struct ViewPort  *View;


SHORT   F[65][65],
/* Das Koordinatennetz */
        Dx[65][65],Dy[65][65];
/* Die Bildschimkoordinaten jedes Punktes */
USHORT  code,i,j;
ULONG   Class;


main ()
{

 printf(">>> 3D Funk-Plotter<<<");
 OpenIntui();
 OpenGfx();
```

```
 Init();
/* Berechnen der Funktion */
 OpenAll(&Window,&Screen,&Rast,&View);
/* Öffnen der Grafikausgabe */
 Ausgabeberechnung();
/* Berechnung der Bildschirmkoordinaten */
 DrawLand();
/* Ausgabe der Funktion */


 /*  Auf Close-Gadget warten und alles schließen  */
 Class = WaitEvent(Window,&code);
 CloseWindow(Window);
 CloseScreen(Screen);
 }


OpenAll(Wind,Scr,RastP,ViewP)

struct Window     **Wind;struct Screen     **Scr;
struct RastPort   **RastP;
struct ViewPort   **ViewP;

{

 /*  Einen low-res Screen öffnen.  */
 *Scr = (struct Screen *)MakeScr(0,0,639,255,
                      "Fractals",2,HIRES,NULL,NULL);
 if (*Scr == NULL)
  exit(FALSE);

  /*  Ein Fenster auf dem neuen Screen öffnen  */
  *Wind = (struct Window *)MakeWindow(0,0,639,255,
          0,0,"Funktionen",WINDOWCLOSE | ACTIVATE,
          CLOSEWINDOW,*Scr);
  if(*Wind == NULL)    /*  Fehler beim Öffnen ?  */
   exit(FALSE);
```

```
 /*  Adresse des Viewports und des Rastports holen */
 *RastP = (*Wind)->RPort;
 *ViewP = &((*((*Wind)->WScreen)).ViewPort);
}


Init()  /* Das Koordinatennetz mit einer Funktion
belegen */

{
   short i,j;

   for (i = 0; i <= 64; i++)
   {
    for (j = 0; j <= 64; j++)
   F[i] [j] = sin(i/16.0*pi) * cos(j/16.0*pi) * 40.0;
   };
}


Ausgabeberechnung ()
/* Berechnet zu den Werten F[x,y] die zugehörigen
                                Bildschirmkoordinaten  */


{
 SHORT    x,y;

 y = 0;
 do {
 x = 0;
 do {
   Dx[x][y] = 512 - (SHORT)(((float)(2*y+192))/32.0*
                 ((float)x)) + 2*y;
   Dy[x][y] = 80 + 2*y - F[x][y];
```

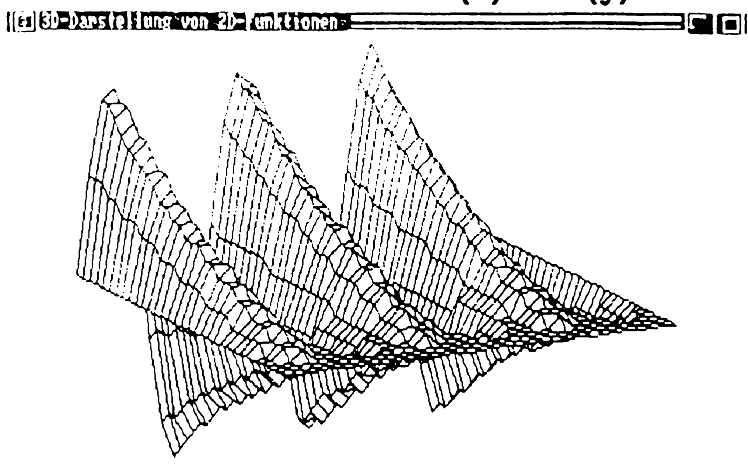
```
    ++x;
  } while (x <= Dim);
  ++y;
 } while (y <= Dim);
}

DrawLand ()
/* Grafische Ausgabe in das Fenster auf dem
                                LoRes-Screen. */

{
  SHORT   x, y;
  struct Coordinates  P[5];

  y = 0;
  do {
   x = 0;
    do {
     P[0].xK = Dx[x][y];
     P[0].yK = Dy[x][y];      P[1].xK = Dx[x+1][y];
     P[1].yK = Dy[x+1][y];
     P[2].xK = Dx[x+1][y+1];
     P[2].yK = Dy[x+1][y+1];
     P[3].xK = Dx[x][y+1];
     P[3].yK = Dy[x][y+1];
     P[4].xK = P[0].xK;
     P[4].yK = P[0].yK;
     Move(Rast,P[0].xK,P[0].yK);
     SetAPen(Rast,0);
     PolyFill(Rast,P,5);
     SetAPen(Rast,1);
     PolyDraw(Rast,5,P);
     ++x;
    } while (x < Dim-1);
    ++y;
  } while (y < Dim-1);
}
```

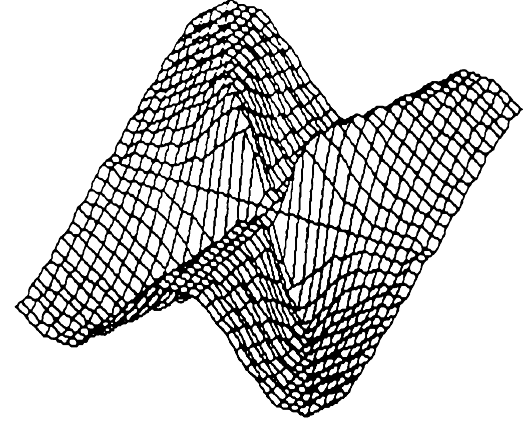
Bild 12.1 - Die Funktion z = sin(x)*sinh(y).



Verwendete Werte:
F[x,y]:=sin(x)*sinh(y) (x:0 bis 6pi,y:0 bis pi/2), wx:=20 Grad, wy:=30 Grad

Bild 12.2 - Noch 'ne Funktion.





Verwendete Werte:
F[x,y] := xy^2/(x^2+y^4) (-2 bis +2), wx := 40 Grad, wy := 60 Grad

Tips

- Auch hier läßt sich optisch durch die Farbgebung viel gewinnen. Benutzen sie doch mal Farbgebungen wie bei den Landschaften aus dem nächsten Abschnitt.
- Bevor Sie eigene Funktionen ausprobieren, machen Sie sich, um Meditationspausen zu umgehen, immer vorher klar, was dabei für Werte herauskommen, bzw. ob überhaupt etwas herauskommt.

5. Künstliche Landschaften

Wir werden Ihnen hier eine einfache Technik vorstellen, mit der Sie mit wenig Rechenzeit eine künstliche Landschaft erzeugen und auf dem Bildschirm darstellen können.

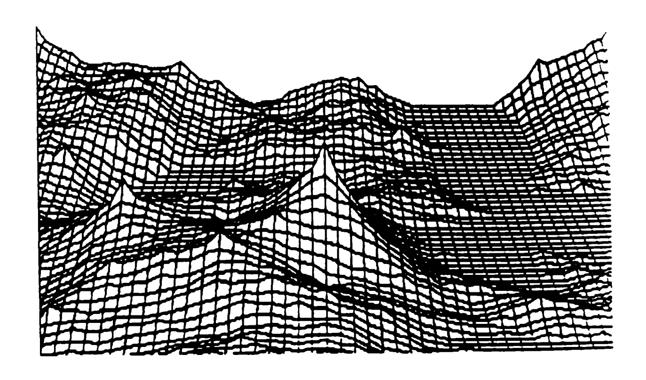
Als Ausgangsfigur benutzen wir dabei wieder das dreidimensionale Netz, das bereits aus dem vorhergehenden Abschnitt bekannt ist. Diesmal wird dieses Netz jedoch nicht mit einer Funktion "belegt", sondern es wird jedem Punkt in diesem Netz eine bestimmte Höhe so zugeordnet, daß eine Landschaft entsteht.

Um die dazu benutzte Technik einfach verständlich zu machen, übertragen wir sie zunächst auf ein zweidimensionales Modell. Diese sind Ihnen gewiß auch aus Atlanten als Landschaftsquerschnitte bekannt. Nehmen wir an, wir wollten einen Querschnitt durch eine zufällige Gebirgslandschaft erzeugen. Eine solche Linie, die die verschiedenen Höhen anzeigt, kann man als Modell wie folgt erhalten: Stellen Sie sich vor, Sie befestigen ein Gummiseil waagerecht auf einer Pinnwand. Haben Sie es nur an den Enden

befestigt, so kann es jetzt in der Mitte auf- und abschwingen. Fassen Sie das Gummi in der Mitte an, ziehen es ein Stück nach oben oder nach unten und befestigen es dort duch eine weitere Nadel. Wiederholen Sie diesen Vorgang mit den neu entstandenden Gummiabschnitten zwischen zwei Nadeln solange bis Sie keine Lust mehr haben, oder der Abstand zwischen zwei Nadeln zu klein wird. - Haben Sie mit dem nach oben oder unten ziehen nicht übertrieben, so müßten Sie jetzt einen mehr oder weniger typischen Landschaftquerschnitt vor sich haben. Die Übertragung dieses Modells auf drei Dimensionen erledigen wir gleich anhand unseres Netzes. Damit wir immer wieder halbieren können, muß die Dimension des Netzes eine Zweierpotenz sein, 64 soll für unsere Zwecke reichen. Wem dies zur Verdeutlichung leichter fällt, kann sich dieses Netz wieder aus einzelnen Gummiseilen zusammengeflochten vorstellen. Das hier vollständig Abgedruckte Programm macht nun nichts anderes als die anfänglich gegebene Fläche der Ausmaße 64x64 in immer kleinere Quadrate zu unterteilen und diese wiederum per Zufall ein kleines Stück nach oben oder nach unten zu ziehen. Bei der Darstellung auf den Monitor kann einerseits wieder die bereits bekannte Prozedur zur Ausgabe eines solchen Netztes verwendet werden, andererseits ist es möglich den Berechnungsvorgang zu beschleunigen, wenn man auf eine variable Darstellung verzichtet. Zur Demonstration verwenden wir diese Technik in unserem Programm "Fraktale Landschaft". Darüberhinaus benutzen wir den bereits aus dem vorherigen Abschnitt bekannten Trick um hinter Erhebungen liegende Flächen zu verdecken. Negative Werte werden ausserdem auf Null gesetzt, wodurch Täler bis zur Höhe Null "aufgefüllt" werden. Durch entsprechende Farbenwahl entstehen so Seen oder Meere. Ansonsten ist auch hier die Farbgebung primitiv gehalten, da dies ein Punkt ist, der die Programmlänge explosionsartig vergrößern kann. Auch hierzu haben wir wieder

einen Bildschirmausdruck (Bild 12.3), der jedoch einen erheblichen Teil seines Reizes verloren hat. Die erzeugten Landschaften entfalten Ihre volle Wirkung nur auf dem (Farb-) Monitor.

Bild 12.3 - Eine zufällig erzeugte Landschaft.



```
/* Landscape */

#include "exec/types.h"
#include"graphics/gfx.h"
#include"intuition/intuition.h"
#include"math.h"
#include"Display.h"
#include"AreaExtras.h"

#define maxDepth  6  /* Die "Berechnungstiefe", legt
auch die Dimension des Koordinatennetzes fest (Dim =
2 Hoch maxDepth) */
```

```
struct Coordinates
        {
         SHORT xK,yK;
        };


struct Window    *Window;
struct Screen    *Screen;
struct RastPort  *Rast;
struct ViewPort  *View;

/* Farbwerte für die einzelnen Farbregister */
int Col []   = { 0,0,2, 2,4,10, 0,5,1, 0,6,1, 1,6,2,
1,7,2, 1,8,2, 2,8,3,
                 2,9,3, 2,10,3, 2,11,3, 3,11,4, 4,11,
4, 4,12,4, 5,9,3, 7,6,0,
                 6,6,1, 5,5,2, 5,4,3, 3,3,3, 4,4,4,
5,5,5, 6,6,6, 7,7,7,
                 8,8,8, 9,9,9, 10,10,10, 11,11,11,
12,12,12, 13,13,13,
                 14,14,14, 15,15,15              };

SHORT    F[65][65],              /* Koordinatennetz */
         Dx[65][65],Dy[65][65],
/* Bildschirmkoordinaten */
         Height[65][65],         /* Durchschnittlicher
Höhenwert eines Polygons */
         Grad[65][65],
/* Steigung eines Polygons */
         Dim,
/* Dimension des Koordinatennetzes */
         Depth,
/* augenblickliche Berechnungstiefe */
         D, d,
/* Hilfsvariablen für die Berchnungstiefe */
         diff, maxh, Net;
float    sigma,delta,            /* Höhenstartwerte */
         Hoeva,                  /* Höhenvarianz */
```

```
        Smooth,                  /* Glättungsfaktor */
        addrnd,pap;
long    seed;                    /* Zufallsgenerator-
Initialisierung */
char    ch;
USHORT  code, i, j, h;
ULONG   Class;


main ()
{

 OpenIntui();
 OpenGfx();

 printf(">>> Programm zur Erstellung fraktaler
Landschaften <<<");
 printf("========================================");
 do {scanf("Seed (lange, beliebige Zahl für den
                        Zufallsgenerator) : %l\n",seed);
 } while( 0 != 0);
 do { scanf(" Hoehe (von 10 bis 100) : %d \n",sigma);
 } while( (sigma <= 10.0)|| (sigma >= 100.0));
 do {scanf(" Hoehenvarianz (von 0.75 bis 2.25) : %d\n
                                            ",Hoeva);
 } while( (Hoeva <= 0.75) || (Hoeva >= 2.25));
 do {scanf(" Glaettungsfaktor (von 0.0 bis 1.0) :
                                      %d\n",Smooth);
 } while( (Smooth <= 0.0) || (Smooth >= 1.0));
 do {scanf(" Ausgabe als >>N<< etz oder >>R<< eal ?
                                          %c \n",ch);
 } while( (ch != 'n') && (ch != 'r')); if (ch == 'n')
  Net = 1;
 else
  Net = 0;

 Init();
 OpenAll(&Window,&Screen,&Rast,&View);
```

```
/* SetColors(Screen,&Col,32);   */

 Depth = 2;
 do {
  Next();               /* Nächste Berechnungstiefe */
  if( Depth > 1 )
  {
   Class = WaitEvent(Window,&code);
   Class = WaitEvent(Window,&code);
   Ausgabeberechnung(); /* Umwandlung in Bildschirm-
                           koordinaten */
   Move(Rast,0,10);
   ClearScreen(Rast);
   DrawLand();           /* Ausgabe der Landschaft */
  };
  ++Depth;
 } while((Depth <= maxDepth));

 /*  Auf Close-Gadget warten und alles schließen  */
 Class = WaitEvent(Window,&code);
 CloseWindow(Window);
 CloseScreen(Screen);
 }


OpenAll(Wind,Scr,RastP,ViewP)

struct Window     **Wind;
struct Screen     **Scr;
struct RastPort   **RastP;
struct ViewPort   **ViewP;
{

 /*  Einen low-res Screen öffnen.  */
 *Scr = (struct Screen *)MakeScr(0,0,320,250,
                          "Fractals",5,NULL,NULL,NULL);
 if (*Scr == NULL)
```

```
  exit(FALSE);

  /* Ein Fenster auf dem neuen Screen öffnen  */
  *Wind = (struct Window *)MakeWindow(0,0,319,250,
           0,0,"Fractals",WINDOWCLOSE | ACTIVATE,
               MOUSEBUTTONS | CLOSEWINDOW,*Scr);
  if(*Wind == NULL)    /* Fehler beim Öffnen ?  */
   exit(FALSE);

/* Adresse des Viewports und des Rastports holen */
*RastP = (*Wind)->RPort;
*ViewP = &((*((*Wind)->WScreen)).ViewPort);
}


SHORT Av3(a,b,c) /* Average3 berchnet den
Durchschnitt dreier Zahlen und
                    addiert einen zufälligen Wert */
SHORT a,b,c;
{
 return ((a+b+c)/3 + (SHORT)(delta * ((rnd()*
2.0-1.0))));
}


SHORT Av4(a,b,c,d)
 /* Average4 berchnet den Durchschnitt vierer Zahlen
    und addiert einen zufälligen Wert */

SHORT a,b,c,d;
{
 return ((a+b+c)/3 + (SHORT)(delta * ((rnd()*
2.0-1.0))));
}

ComputePos(a,b)    /* Berechnet die Zwischenpunkte
                                       (3.Stufe) */
```

```
SHORT a,b;
{
 SHORT x, y, hx, hy;
 x = a;
 hx = y+d;
 while((x <= Dim-d) && (hy <= Dim))
 {
  y = b;
  hy = y+d;
  while ((y <= Dim-d) && (hy <= Dim))
  {
   F[x][y] = Av4(F[x] [hy],F[x] [y-d],F[hx] [y],
   y += D;
   hy = y+d;                                  F[x-d] [y])
  };
  x += D;
  hx = x+d;
 };
}


AddRandom(a,b)  /*  Addiert zu den  Höhenwerten  einen
zufälligen Wert */

SHORT a,b;
{
 SHORT x,y;
 x =a;
 do {
  y =a;
  do {
   F[x] [y] =F[x] [y] + (delta * ((rnd()*2.0-1.0)));
   y += D;
  } while (y <= b);
  x += D;
 } while (x <= b);
}
```

```
Init()   /* Initialisierung der Eckpunkte */

{
 SHORT t;
 float f;
 Dim = PowerInt(2,maxDepth);
 t = 0;
 delta = sigma;
 t = (delta * rnd());
 F[0] [0] = t + 50.01;
 t = (delta * (rnd()*2.0-1.0));
 F[0] [Dim] = t + 20.01;
 f = rnd();
 t = (delta * f);
 F[Dim] [0] = t + 80.01;
 t = (delta * (0.0-rnd()));
 F[Dim] [Dim] = t + 60.1;
 D = Dim / 2;
 t = (delta * (rnd()*2.0-1.0));
 F[D] [D] = t + 70.01;
 d = D / 2;
}


Next()  /* Nächste Berechnungstiefe */

{
  SHORT x, y;

  delta = delta * Pow(0.5,Hoeva);  /* Höhenwert delta
                                        verkleinern */

/* Erste Stufe der Berechnung: Alle Punkte ermitteln
die diagonal zwischen vier bereits bekannten Höhen-
werte liegen. */
```

```
  x = d;  do {
   y = d;
   do {
    F[x] [y] = Av4(F[x][y+d],F[x+d][y-d],F[x-d]
[y+d],F[x-d][y-d]);
    y += D;
   } while (!(y > Dim-d));
   x += D;
  } while (!(x > Dim-d));

  addrnd = rnd();
  if (addrnd > Smooth)
   AddRandom(0,Dim);

  delta = delta * Pow(0.5,Hoeva);
/* Höhenwert delta verkleinern */

/* Zweite Stufe der Berechnung: Alle Randpunkte er-
mitteln, die zwischen drei bereits bekannten Höhen-
werte liegen. */

  x = d;
  do
  {
   F[x] [0] = Av3(F[x+d] [0],F[x-d] [0],F[x] [d]);
   F[x] [Dim] = Av3(F[x+d] [Dim],F[x-d] [Dim],F[x]
[Dim-d]);
   F[0] [x] = Av3(F[0] [x+d],F[0][x-d],F[d][x]);
   F[Dim][x] = Av3(F[Dim] [x+d],F[Dim] [x-d],F[Dim-d]
[x]);
   x += D;
  } while (!(x > Dim-d));

/* Dritte Stufe der Berechnung: Alle noch übrigge-
bliebenen Zwischenpunkte ermitteln. */

  ComputePos(d,D);
  ComputePos(D,d);
```

```
  addrnd = rnd();
  if (addrnd > Smooth)
  {
   AddRandom(0,Dim);   AddRandom(d,Dim-d);
  };

  D = D / 2;  /* Hilfsvariablen auf den nächsten Ver-
                 kleinerungsschritt */
  d = d / 2;  /* vorbereiten. */

}


Ausgabeberechnung ()
/* Berechnet zu den Werten F[x,y] die zugehörigen x-
und y-Koordinaten  */
/* sowie die Höhe und die Steigung der einzelnen
Polygone.               */


{
 SHORT    x,y,yi;
 float  yr;
 y = 0;
 do {
  x = 0;
  do {
   Dx[x][y] = 256 - (SHORT)(((float)(2*y+192))/
64.0*((float)x)) + y;
   if (F[x][y] <= 0)          /* Punkt liegt unter dem
                                 Meeresspiegel */
    Dy[x][y] = 111 + 2*y;
   else                       /* Punkt liegt über dem
                                 Meeresspiegel */
    Dy[x][y] = 111 + 2*y - F[x][y];
   x += D;
  } while (x <= Dim);
```

```
  y += D;
 } while (y <= Dim);

 maxh = 29;

 y = 0;
 do {
  x = 0;
  do {
/* Durchschnittshöhe der Polygone berechnen */
Height[x][y] = (F[x][y]+F[x][y+D]+F[x+D][y+D]+F[x+D]
[y])/4;

    if ( ( (F[x][y]>0) || (F[x][y+D]>0) || (F[x+D]
[y]>0) || (F[x+D][y+D]>0) ) && (Height[x][y]<=0) )

/* Meerespiegel für dieses Polygon nur setzen, wenn
alle Eckpunkte im "Wasser" liegen. */
     Height[x][y] = 1;

/* Steigung des Polygons bestimmen */
    Grad[x][y] = abs(abs(F[x][y]) + abs(F[x+D][y]) -
                 2*Height[x][y]) / D;
    if (Grad[x][y] > 7)
     Grad[x][y] = 7;
    if (maxh < Height[x][y])
     maxh = Height[x][y];
    x += D;
  } while (x <= (Dim-D));
  y += D;
 } while (y <= (Dim-D));
}
```

```
DrawLand ()
/* Grafische Ausgabe in das Fenster auf dem LoRes-
Screen. */

{
  SHORT   x, y, c, s;
  float dh, ds, h;
  struct Coordinates  P[5];

  dh = ((float) maxh)/29.0;
  y = 0;
  do {
   x = 0;
    do {
     P[0].xK = Dx[x][y];
     P[0].yK = Dy[x][y];
     P[1].xK = Dx[x+D][y];
     P[1].yK = Dy[x+D][y];
     P[2].xK = Dx[x+D][y+D];
     P[2].yK = Dy[x+D][y+D];
     P[3].xK = Dx[x][y+D];
     P[3].yK = Dy[x][y+D];
     P[4].xK = P[0].xK;
     P[4].yK = P[0].yK;
/* Farbbestimmung, dazu in h einen Höhenwert von
                                        maximal 30 */
     h = ((float)Height[x][y])/dh;
     s = Grad[x][y];
     if (h <= 0.0)    /* H;he <= 0, also Meer(blau) */
      c = 1;
     else
     if (h > 24.0)    /* Höhe > 24, also Steingrau bis
                         Schneeweis */
      c = (2.5 + h);
     else
     if (s > 3)       /* Steigung > 50 Grad, Also
                         Felsgrau */
      c = 27 - s;
```

```
    else              /* sonst "normale" Höhenlinie */
      c = 2.5 + h;


    Move(Rast,P[0].xK,P[0].yK);
    SetAPen(Rast,0);
    PolyFill(Rast,P,5);
    SetAPen(Rast,c);
    PolyDraw(Rast,5,P);

    x += D;
   } while (x <= Dim-D);
   y += D;
  } while (y <= Dim-D);
}
```

Tips:

- Die so erzeugten Grafiken lassen sich um ein vielfaches verschönern, indem Sie mehr Farben einsetzten und die Auflösung erhöhen, also das 64x64 große Netz durch ein 128x128 oder gar 256x256 großes ersetzen. Dabei kann es allerdings Probleme mit dem Compiler geben, da ein doppelt dimensioniertes Feld nicht beliebig groß werden kann. Eventuell müssen Sie ein großes Netz aus mehreren 64x64 großen Feldern zusammensetzten.
- Lassen Sie uns noch folgende Anregungen zur Farbgebung geben. Im Normalfall setzt sich die Landschaft aus 64x64 Feldern zusammen. Von jedem dieser Felder kennen wir die vier Höhenwerte der Eckpunkte. Daraus läßt sich nicht nur die Durchschnittshöhe jedes einzelnen Feldes errechnen, sondern auch Steigung, bzw. Neigung um die x- und yAchse. Entsprechend der Durchschnittshöhe setzt man Farben wie Meeresblau für die Höhe Null, verschiedene Grüntöne für die folgenden Höhen, sowie über einige

Grautöne, schließlich weiß für den Schnee auf den Bergspitzen. Ab einer gewissen Steigung können Sie verschieden Grautöne verwenden um zu verdeutlichen, daß es sich um felsige Gebiete handelt.
- Um auch eine gewisse Schattierung hereinzubringen, kann man die Neigung zum Betrachter oder einer imaginären Lichtquelle berechnen. Je stärker sich ein Feld von diesen Werten neigt, desto schwächer wird Licht reflektiert, also um so dunkler muß dieses Feld erscheinen.

6. Kugeln mit Längen- und Breitengrade

Abschließend befassen wir uns mit der Darstellung einer Kugeloberfläche mit Hilfe von Längen- und Breitengraden. Es entsteht also lediglich ein Gittermodell einer Kugel, dessen Auflösung wir dadurch erhöhen, daß wir sie nicht aus einzelnen Linien, sondern aus einzelnen Punkten zusammensetzen. Dieses Verfahren ist zwar etwas zeitaufwendiger, aber setzt man die Kugel aus nur wenigen Längen- und Breitengraden zusammen, so erscheint bereits nach wenigen Minuten die fertige Kugel auf dem Bildschirm. Zur Erstellung betrachten wir zunächst nur die Längengrade. In unserem definierten Koordinatensystem sind dies Kreise, die alle den gleichen Radius haben, nämlich den der Kugel. Der Mittelpunkt eines jeden Kreises befindet sich im Ursprung des Koordinatensystems. Den Nullmeridian legen wir in die xy-Ebene, er erscheint also auf dem Bildschirm als Kreis. Dieser Kreis kann mit Hilfe der Rotationsmatrix wie folgt berechnet werden: die Schnittpunkte des Kreises mit der x-Achse sind uns durch den Radius bekannt. Drehen wir nun mit Hilfe der Rotationsmatrix einen dieser Punkte

schrittweise um die z-Achse, so erhalten wir die Punkte, die auf dem Kreis liegen. Der erste Meridian wäre somit berechnet. Die anderen erhalten wir, wenn wir diesen Meridian Schrittweise um die y-Achse drehen.

Die Breitengrade dagegen haben nicht alle den gleichen Radius. Der Äquator hat den größten Radius, den der Kugel, während die anderen zu den Polen hin immer kleiner werden. Diese Kreise sind alle parallel zu der xz-Ebene und ihre Mittelpunkte liegen alle auf der y-Achse, der des Äquators im Ursprung des Koordinatensystems. Diesmal liegt unser Ausgangspunkt zur Berechnung der Kreise nicht auf einer der Koordinatenachsen, sondern auf dem Nullmeridian. Tasten wir diesen Schrittweise ab und drehen diese Punkte um die y-Achse, so erhalten wir bereits die gewollten Breitenkreise.

In der Praxis ist es über die Rotationsmatrix möglich diese Kugel auch noch beliebig zu drehen, indem wir einfach die gewünschten Drehwinkel wie bekannt in die Rotationsmatrix einsetzen. Dabei ist es hier ein Leichtes, die hintenliegenden Punkte auszurechnen und gar nicht erst zu zeichnen. Bei unserer Kugel sind alle Punkte unsichtbar, die hinter der xy-Ebene liegen, also negative z-Koordinaten haben. Hat also ein Punkt eine z-Koordinate kleiner als Null, so wird er nicht gesetzt. Der wesentliche Programmteil sieht so aus:

```
/* Kugel                                    von Olaf Pfeiffer,
                            C-Version von Paul Lukowicz   */

#include"Display.h"
#include"graphics/gfx.h"
#include"intuition/intuition.h"
```

```
#include"math.h"
#include"stdio.h"

#define  pi     3.14
#define  wx     0.62    /* Drehung um die x-Achse   */
#define  wz     0.39    /* Drehung um die z-Achse   */
#define  wy     0.0     /* Drehung um die y-Achse
                                    (uninteressant) */
#define  Radius   75.0
                /* Kugelradius in Pixeln (10 - 100) */
#define  Laengen  12.0
                /* Anzahl der Längengerade (2 - 32) */
#define  Breiten  11.0
                /* Anzahl der Breitengerade (1 - 31)*/

struct Window     *Window;
struct Screen     *Screen;
struct RastPort   *Rast;
struct ViewPort   *View;

float  s;
float  M [5] [5];

int Col []   = { 0, 0, 2,  2, 4,10,  0, 5, 1,  0,
6, 1,  1, 6, 2,  1, 7, 2,
                 1, 8, 2,  2, 8, 3,  2, 9, 3,  2,
10, 3,  2,11, 3,  3,11, 4,
                 4,11, 4,  4,12, 4,  5, 9, 3,  7,
6, 0,  6, 6, 1,  5, 5, 2,
                 5, 4, 3,  3, 3, 3,  4, 4, 4,  5,
5, 5,  6, 6, 6,  7, 7, 7,
                 8, 8, 8,  9, 9, 9, 10,10,10, 11,
11,11, 12,12,12, 13,13,13,                 14,14,
14, 15,15,15
               };
```

```
main ()
{
  USHORT  code, i, j, h;
  ULONG   Class;

  OpenIntui();
  OpenGfx();

  /*  Einen low-res Screen öffnen.  */
  Screen = (struct Screen *)MakeScr(0,0,320,250,
            "HAM-Demo",5,NULL,NULL,NULL);
  if (Screen == NULL)
   exit(FALSE);

  /*  Ein Fenster auf dem neuen Screen öffnen  */
  Window = (struct Window *)MakeWindow(0,0,320,
            250,0,0,"Ham-Demo",
            WINDOWCLOSE | ACTIVATE | WINDOWDRAG,
            CLOSEWINDOW,Screen);
  if(Window == NULL)    /*  Fehler beim Öffnen ?  */
   exit(FALSE);

  /*  Adresse des Viewports und des Rastports holen,
                                   Farben setzten  */
  Rast = (Window)->RPort;
  View = &((*(Window->WScreen)).ViewPort);
  /*for(i = 0; i<32;SetRGB4(View,(i+=3),Col[i-3],Col
[i-2],Col[i-1]));*/
  s = 1.0 / Radius;
  Matrix ();
  LCircle(Laengen);
  BCircle(Breiten);

  /*  Auf Close-Gadget warten und alles schließen  */
  Class = WaitEvent(Window,&code);
  CloseWindow(Window);
  CloseScreen(Screen);
}
```

```
/* Belegen der Rotationsmatrix */

Matrix ()

{
   M[1] [1] = cos(wz) * cos(wy);
   M[2] [1] =-cos(wz) * sin(wy);
   M[3] [1] = sin(wz);
   M[1] [2] = cos(wx) * sin(wy) + sin(wx)
                 * cos(wy) * sin(wz);
   M[2] [2] = cos(wx) * cos(wy) - sin(wx)
                 * sin(wy) * sin(wz);
   M[3] [2] =-sin(wx) * cos(wz);
   M[1] [3] = sin(wx) * sin(wy) - cos(wx)
                 * cos(wy) * sin(wz);
   M[2] [3] = sin(wx) * cos(wy) + cos(wx)
                 * sin(wy) * sin(wz);
   M[3] [3] = cos(wx) * cos(wz);
}

Drawxyz(p, q)

float p,q;

{
   float x,y,z,x1,y1,z1;
   int   Posx,Posy,c;

   x = cos(q) * cos(p);
   y = cos(q) * sin(p);
   z = sin(q);
   x1 = M[1][1] * x + M[2][1] * y + M[3][1] * z;
   y1 = M[1][2] * x + M[2][2] * y + M[3][2] * z;
   z1 = M[1][3] * x + M[2][3] * y + M[3][3] * z;
  if( y1 >= 0.0)
  {
     Posx = 160 + (int)(Radius * x1);
     Posy = 130 + (int)(Radius * z1);
```

```
      c = (int)(y1 * 30.0) + 2;
      SetAPen(Rast,c);
      WritePixel(Rast,Posx,Posy);
   };
}


LCircle (step)

float step;

{
   float p, q, st,pi2;

   st = pi/step;
   pi2 = 2.0 * pi;
   for(p = 0.0; p< pi; p= p+st)
    for(q = 0.0; q < pi2;Drawxyz(p,q), q = q +s);
}

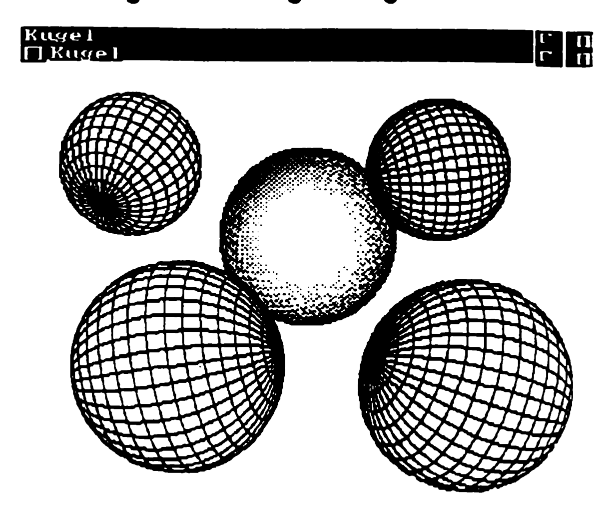
BCircle(step)

float step;

{
   float p,q,st,pi2,pi3;

   pi2 = pi / 2.0;
   pi3 = pi * 2.0;
   st = pi / (step +1);
   for(q = (-pi2 + st);q <= (pi2 - st);p = p)
   {
     for(p = 0.0; p < pi3;Drawxyz((p),q),p = p+s);
q = q + st;
   };
}
```

Bild 12.4 - Kollage von mehreren durch das Programm "Kugel" erzeugte Kugeln.



Tips:

- Wiedereinmal zur Farbgebung: in diesem Beispiel ist die Farbe abhängig von der Tiefe, von der z-Koordinate eines Punktes. Wählt man die Farben entsprechend, entsteht der Eindruck unser Modell würde direkt von vorne beleuchtet. Wollen Sie eine seitliche Beleuchtungsquelle simulieren, so müssen Sie die Farbgebung winkelabhängig machen. Von jedem Punkt kennen Sie durch die Koordinaten auch die Winkel, in denen er zu den einzelnen Achsen steht. Setzen Sie beispielsweise den Punkt der höchsten Reflexion, also der Punkt der auf der Kugeloberfläche am hellsten erscheint, so, daß er zu allen drei Achsen im Winkel Pi/4 steht. Je größer nun der Unterschied der drei Winkel eines Punktes zu dem

gesetzten Wert (Pi/4) ist, desto dunkler muß dieser Punkt werden.

7. Abschließende Anregungen für Fortgeschrittene

Wir haben Ihnen nun viele verschieden Techniken vorgestellt, mit denen Sie innerhalb kurzer Zeit zum Teil bereits recht verblüffende Computergrafiken erstellen können. Für diejenigen unter Ihnen, die bereits eigene Änderungen, Erweiterungen oder Verbesserungen an diesen zum Teil recht einfachen Grafikalgorithmen vorgenommen haben noch ein Vorschlag: versuchen Sie die Techniken miteinander zu verknüpfen! Erstellen Sie doch zum Beispiel einen Landschaftsauschnitt und "bepflanzen" Sie diesen mit verschiedenen Gewächsen der L-Systeme, womöglich auch noch in Abhängigkeit von der jeweiligen Höhe.

Oder eine andere Idee: schreiben Sie das Programm zur Erstellung der Kugeln so um, daß nicht mehr Punkt für Punkt sondern nur noch Schnittpunkt für Schnittpunkt (von Längen- und Breitengrad) berechnet wird und diese dann durch Linien verbunden werden. So ist es nicht mehr weit bis zu dem Punkt, wo Sie das Netz der fraktalen Landschaft auf diese Kugel projezieren können. Somit hätten Sie dann ein Programm zur Erstellung Ihrer eigenen Planeten oder, je nach Farbgebung und Höhenvarianz, Monde und Kometen.

Anhang A

==
Die Datenstrukturen der Intuition
==

Dieser Teil des Anhangs enthält die wichtigsten Datenstrukturen der Intuition in alphabetischer Reihenfolge. Dabei haben wir vor allem die ausgewählt, die für die Programmierung und insbesondere die Grafikprogrammierung wichtig sind. Da einige der Datenstrukturen relativ lang sind, haben wir uns bei den Datenstrukturen **Window** und **Preferences** auf die wesentlichen Teile beschränkt.

Border

Diese Struktur wird von der Routine **DrawBorder** verwendet, um einen beliebigen Rahmen zu zeichnen. Sie beinhaltet alle Informationen, die **Draw-Border** für das Zeichnen der Umrandung braucht. Sie können über **NextBorder** mehrere **Border**-Strukturen verketten, um sie mit einem **DrawBorder**-Aufruf ausgeben zu können.

```
struct Border
{
  SHORT  LeftEdge, TopEdge;
  UBYTE  FrontPen, BackPen, DrawMode;
  BYTE   Count;J
  struct Border *NextBorder;
};
```

LeftEdge, TopEdge: Hierbei handelt es sich um die zum Rastport relative Position des Rahmens (x- und y-Koordinate in Pixels).

328

FrontPen: Die Nummer des Farbregisters mit dem Farbwert, der als Vordergrundfarbe beim Zeichnen des Rahmens benutzt werden soll.

BackPen: Die Hintergrundfarbe.

DrawMode: Der Zeichenmodus, in dem der Rahmen gezeichnet werden soll. (Siehe Prozedur **Draw**)

Count: Anzahl der XY-Koordinatenpaare, die die Eckpunkte des Rahmens bilden.

***XY:** Die Koordinatenpaare relativ zur linken oberen Ecke. Sie werden in einem Speicherbereich, auf den **XY** zeigt, abgelegt.

***NextBorder:** Zeiger auf die nächste **Border**-Struktur, die ebenfalls noch ausgeführt werden soll.

Ist dieser Wert gleich **NULL**, so wird keine weitere Umrandug gezeichnet.

Image

Dies ist eine kurze Struktur für die einfache Übertragung von Grafikausschnitten zu einem RastPort mit Hilfe der ***DrawImage***-Prozedur. Zusätzlich bietet diese Struktur über **PlanePick** und **PlaneOnOff** die Möglichkeit, dabei die Daten in ausgewählte Bitplanes des Screens zu schreiben.

```
struct Image
{
  SHORT   LeftEdge,TopEdge, Width, Height, Depth;
  USHORT  *ImageData;
  UBYTE   PlanePick, PlaneOnOff;
  struct  Image *NextImage;
};
```

LeftEdge, TopEdge: Position der linken oberen Ecke des Bildausschnitts relativ zum Rastport (x- und y-Koordinaten in Pixels).

Width, Height: Die Breite und Höhe des Ausschnitts in Pixel.

Depth: Die Tiefe der Grafik, also die Anzahl der Bitplanes.

***ImageData**: Ein Zeiger auf den Anfang der Grafikdaten im Speicher (**muß CHIP-RAM sein**) Die Datem sind Bitplaneweise hintereinander gespeichert.

PlanePick: Hier geben Sie an, in welche Bitplanes des Rastports die Bildaten des Images geschrieben werden sollen.

PlaneOnOff: Gibt an, welche Farbe den Punkten zugeordnet wird, die in den Imagedaten den Wert **NULL** haben.

***NextImage**: Zeiger auf die nächste **Image**-Struktur, die auch noch ausgegeben werden soll. Ist dieser Wert gleich **NULL**, so wird kein weiteres Image gezeichnet.

IntuiMessage

Diese Struktur wird von Intuition benutzt, um Systemnachrichten über Ereignisse in einem Fenster zu übermitteln.

```
struct IntuiMessage;
{
  struct  Message ExecMessage;
  ULONG   Class;
  USHORT  Code,Qualifier;
  APTR    IAddress;
  SHORT   MouseX, MouseY;
  ULONG   Seconds, Micros;
  struct  Window *IDCMPWindow;
  struct  IntuiMessage *SpecialLink;
};
```

Class: Dieses Feld enthält die **IDCMPFlags** der in einem Fenster aufgetretenen IO-Ereignisse, oder **NULL** falls keine Ereignisse in dem Fenster registrert wurden, zu dem die Struktur gehört. Zur Bedeutung der Flags siehe unten.

Code: In diesem Feld werden weitere Werte, wie z.B. die Menü-Nummern oder der ASCII-Code der gedrückten Taste.

Qualifier: Dies ist eine Kopie des aktuellen Eingabe-Qualifiers (z.B. gedrückte ALT-Taste).

IAddress: Ein Zeiger auf die Adresse des Intuition-Gebildes, das diese Nachricht auslöste, z.B. ein Zeiger auf eine Gadget oder Requester-Struktur.

MouseX, MouseY: Dies sind die x-und y-Koordinaten der Mausposition (relativ zur linken oberen Ecke des aktuellen Fensters) zur Zeit des Eintreffens der Nachricht.

Seconds, Micros: Die Systemzeit in Sekunden und Mikrosekunden, zu der die Nachricht geschickt wurde.

***IDCMPWindow:** Zeiger auf das Fenster, zu dem diese Struktur gehört.

Folgende **IntuiMessage- IDCMPFlags** können abgefragt werden (In der dahinter stehenden Bemerkung wird kurz erklärt, was es bedeutet, wenn dieses Bit gesetzt ist):

SIZEVERIFY 0x00000001
Der Benutzer will die Fenstergröße verändern.

NEWSWIZE 0x00000002
Die Größe des Fensters hat sich verändert.

REFRESHWINDOW 0x00000004
Die Refresh-Routine soll aufgerufen werden.

MOUSEBUTTONS 0x00000008
Ein Mausknopf wurde betätigt. In Code kann stehen:

SELECTDOWN, SELECTUP: Die linke Taste wurde gedrückt, bzw. wieder losgelassen.
MENUDOWN, MENUUP: Analog für die rechte Taste.

MOUSEMOVE 0x00000010
Die Maus wurde bewegt.

GADGETDOWN 0x00000020
Die Maustaste wird über einen Gadget "gedrückt" gehalten. Die Adresse des Gadgets finden Sie in **IAddress**.

GADGETUP 0x00000040
Die Maustaste wurde über einem Gadget "losgelassen". Die Adresse des Gadgets finden Sie in **IAddress**.

REQSET 0x00000080
Ein Requester wurde aktiviert.

MENUPICK 0x00000100
Ein Menüpunkt ist ausgewählt worden. Die Menü-Nummer (in der Form eines Intuition **MenuNumber**) finden Sie in Code.

CLOSEWINDOW 0x00000200
Das Close-Gadget wurde angeklickt.

RAWKEY 0x00000400
Eine Taste wurde gedrückt. **Code** enthält die Nummer der Taste. Zusätzliche Information wie z.B. gedrückte Alt- oder Amiga-Taste in **Qualifier**.

REQVERIFY 0x00000800
Der Benutzer versucht ein Requester zu aktivieren.

REQCLEAR 0x00001000
Der letzte Requester vom Screen wurde gelöscht.

MENUVERIFY 0x00002000
Der Benutzer versucht ein Menü zu aktivieren.

NEWPREFS 0x00004000
Neue Preferences wurden eingelesen.

DISKINSERTED 0x00008000
Eine neue Diskette wurde eingelegt.

DISKREMOVED 0x00010000
Eine Diskette wurde herausgennomen.

WBENCHMESSAGE 0x00020000
Es ist eine Nachricht der Workbench vorhanden.

ACTIVEWWINDOW 0x00040000
Das Fenster wurde aktiviert (angeklickt).

INACTIVEWWINDOW 0x00080000
Das Fenster wurde inaktiviert (Ein anderes also angeklickt) !

DELTAMOVE 0x00100000
Die Maus wurde bewegt. Die angegebenen Mauskoordinaten sind hier relative Koordinaten.

VANILLAKEY 0x00200000
Eine Taste wurde gedrückt. Der Tastencode wurde in ASCII-Codes umgewandelt und liegt in **Code** vor. (Falls eine Taste gedrückt wurde).

INTUITICKS 0x00400000
Es liegt eine Timer-Nachricht vor.

LONELYMESSAGE 0x80000000
Ist dieses Bit gelöscht, so ist es eine Systemnachricht an einen Task, sonst eine von einem Task.

IntuiText

Die Datenstruktur **IntuiText** ermöglicht die Ausgabe einer oder mehrerer Zeichenketten, mit der Ausgaberoutine **PrintIText**. Dabei können etliche Eigenschaften des Textes unabhängig von den Einstellungen des Rastports bestimmt werden.

```
struct IntuiText
{
  UBYTE  FrontPen, BackPen, DrawMode;
  SHORT  LeftEdge, TopEdge;
  struct TextAttr *ITextFont;
  UBYTE  *IText;
  struct IntuiText *NextText;
};
```

FrontPen, BackPen: Die Nummer der Farbregister, deren Farbwert als Vordergrund-, bzw. Hintergrundfarbe für den Text benutzt wird.

DrawMode: Der Zeichenmodus, in dem geschrieben werden soll.

Leftedge, TopEdge: Die zum aktuellen Fenster relative x- und y-Koordinate, an der die linke, obere Ecke des Textes stehen soll.

ITextFont: Dies ist der Zeiger auf den Zeichensatz (Eigentlich **TextAttr**-Struktur), der für die Textausgabe benutzt werden soll. Ist dieser gleich **NULL** gesetzt, so wird der aktuelle Zeichensatz verwendet.

***IText:** Zeiger auf den auszugebenen Textstring, der mit einer Null enden muß.

***NextText:** Zeiger auf die nächste **IntuiText**-Struktur, die noch ausgegeben werden soll, oder **NULL** für Ende.

NewScreen

Diese Struktur wird jedesmal benutzt, wenn ein neuer Screen geöffnet wird. Beachten Sie auch die Struktur **Screen**, die nach dem Öffnen weitere Informationen eines Screens enthält.

```
struct NewScreen
{
  SHORT  LeftEdge, TopEdge, Width, Height, Depth;
  UBYTE  DetailPen, BlockPen;
  USHORT ViewModes, Type;
  struct TextAttr *Font;
  UBYTE  *DefaultTitle;
  struct Gadget *Gadgets;
  struct BitMap *CustomBitMap;
};
```

LeftEdge, TopEdge: Die x- und y-Koordinaten der linken, oberen Ecke der Screenbegrenzung, wobei **LeftEdge** nicht benutzt wird, da der Screen immer am linken Bildschirmrand beginnt.

Width, Height: Die Gesamtbreite und -höhe des Screens in Pixel.

Depth: Die Tiefe des Screens, also die Anzahl der verwendetete Bitmaps. Im allgemeinen stehen dann 2 hoch dieser Zahl Farben in diesem Screen zur Verfügung.

DetailPen: Die Nummer des Farbregisters, das den Farbwert enthält, mit dem der "Zeichenstift" in diesem Screen zeichnen, bzw. schreiben soll.

BlockPen: Die Nummer des Farbegisters, das den Farbwert enthält, mit dem die Menüzeile mit dem Titel und den Gadgets gezeichnet werden soll.

ViewModes: Bestimmt die Anzeigemodi des neuen Screens. Sie können hier einen oder mehrere folgender Flags übergeben:

> **HIRES:**
> High-Resolution Modus an (640 Pixels horizontal).
>
> **INTERLACE:**
> Interlacemodus an (max 512 Bildzeilen).
>
> **DUALPF:**
> Dual Playfields Modus an
>
> **HAM:**
> Hold-And-Modify Modus an.

SPRITES:
Sprites können auf dem Screen dargestellt werden.

Wird **HIRES** und/oder **INTERLACE** nicht gesetzt, dann wird ein **LowRes** (320 Pixel horizontal) und/oder Normalmodus eingeschaltet.

Type: Dieser Wert gibt an, ob es sich um einen Workbench-Screen oder Custom-Screen handelt.

***Font:** Dies ist ein Zeiger auf den Zeichensatz, der innerhalb dieses Screens verwendet werden soll.

***DefaultTitle:** Ein Zeiger auf den Textstring, der den Titel dieses Screens beinhaltet. Dieser erscheint dann in der Menüleiste.

***Gadgets:** Dies ist ein nicht benutzter Zeiger, den Sie auf **NULL** setzten sollten.

***CustomBitMap:** Öffnen Sie einen Custom-Screen (und nicht ein Workbench-Screen),so haben Sie die Möglichkeit,eine eigene Bitmap zu verwenden. Setzen Sie dazu in Types die **CUSTOMBITMAP**-Flags und stellen Sie diesen Zeiger auf die **BitMap**-Struktur dieser Bitmap.

NewWindow

Wird ein neues Fenster auf einem Screen geöffnet, so wird diese Struktur benutzt, um die Daten des neuen Fensters an **OpenWindow** zu übergeben. Dabei ist die Struktur **Window** wieder diejenige, die nach dem Öffnen weitere Informationen über ein Fenster enthält.

```
struct NewWindow
{
  SHORT  LeftEdge, TopEdge, Width, Height;J
  ULONG  IDCMPFlags, Flags;
  struct Gadget *FirstGadget;
  struct Image *CheckMark;
  UBYTE  *TITLE
  struct Screen *Screen
  struct BitMap *BitMap;
  SHORT  MinWidth, MinHeight, MaxWidth, MaxHeight;
  USHORT Type;
};
```

LeftEdge, TopEdge: Die x- und y-Koordinaten der linken oberen Ecke des Fensters in Pixels relativ zum Screen.

Width, Height: Die Gesamtbreite und -höhe des Fensters in Pixels.

DetailPen: Die Nummer des Farbregisters, die den Farbwert enthält, mit dem der imaginäre Zeichenstift in diesem Fenster zeichnen, bzw. schreiben soll.

BlockPen: Die Nummer des Farbegisters, die den Farbwert enthält, mit dem die Menüzeile mit dem Titel und den Gadgets, sowie der Rahmen gezeichnet werden soll.

IDCMPFlags: Hier geben Sie die IDCMP-Flags, die gleich beim Öffnen des Fensters gesetzt werden sollen, an. Zur Bedeutung der Flags siehe **IDCMPMessage** und **Window.**

Flags: Die **WindowFlags**, die einige wichtige Eigenschaften des Fensters bestimmen. Das Setzen eines Flags veranlaßt **OpenWindow** dazu, das Fenster beim Öffnen mit der entsprechenden Eigenschaft auszustatten.

WINDOWSIZING: In dem Fenster erscheint unten links das Sizing-Gadget, welches dem Benutzer erlaubt,die Größe des Fensters zu verändern. Dabei können Sie durch die Flags SIZE(B)RIGHT oder SIZEBOTTOM bestimmen, ob das Gadget dem rechten, dem unteren oder beiden Rändern zugeordnet werden soll.

WINDOWDEPTH: Mit diesem Flag ermöglichen Sie es dem Anwender, das Fenster in den Vorder- oder Hintergrund zu schalten, denn die dafür verantwortlichen Gadgets erscheinen in der oberen rechten Ecke des Fensters.

WINDOWCLOSE: Das Close-Gadget zum Schließen des Fensters erscheint in der linken oberen Ecke des Fensters. Wird es vom Benutzer angeklickt, so erhalten Sie von der Intuition eine entsprechende Message.

WINDOWDRAG: Haben Sie dieses Flag gesetzt, so kann der Benutzer das Fenster durch Ziehen der Titelleiste verschieben.

GIMMEZEROZERO: Aktiviert ein GZZ-Fenster, d.h. der Koordinaten-Nullpunkt liegt in der inneren - oberen -linken Ecke des Fensters.

SIMPLE_REFRESH: Wird ein Teil des Fensters von einem anderen enthüllt, so muß der Inhalt neu aufgebaut werden.

SMART_REFRESH: Der Inhalt des Fensters muß nur neu aufgebaut werden, wenn das Fenster vergrößert wird.

SUPER_BITMAP: Setzen Sie dieses Bit, um ein **SuperBitmap**-Window zu erhalten. Vergessen Sie nicht, daß **BITMAP** dabei auf Ihre eigene **BitMap**-Struktur zeigen muß.
BACKDROP: Aktiviert das Fenster als ein Backdrop-Fenster.

REPORTMOUSE: Das Fenster erhält die Mauszeigerbewegungen als x- und y-Koordinaten.

BORDERLESS: Das Fenster erscheint ohne Rahmen. Beachten Sie dabei, daß jedoch Rahmenteile erscheinen können, wenn Sie System-Gadgets oder einen Titeltext definiert haben.

ACTIVATE: Das Fenster wird beim Öffnen auch automatisch aktiviert. Eine logische Folge ist die inaktivierung eines eventuell vorhandenen aktiven Fensters, daß möglicherweise gerade zur Tastatureingabe benutzt wird. Es kann also passieren, daß Sie dem Anwender "den Boden unter den Füßen wegziehen", indem Sie ihm die Kontrolle über sein Fenster klauen.

NOCAREREFRESH: Setzen Sie dieses Bit, wenn Sie keine Refresh-Messages von der Intuition erhalten wollen.

RMBTRAP: Durch Setzen dieses Bits legen Sie fest, daß zu diesem Fenster keine Menüleiste definiert ist. Sie erhalten jedoch weiterhin die **MouseButtonEvents**.

FirstGadget:** Dies ist ein Zeiger auf eine **Gadget**-Struktur, mit der Sie die zu diesem Fenster gehörenden Gadgets definieren können. ***Achtung! die System-Gadgets werden durch **Flags** aktiviert.

***CheckMark:** Dies ist ein Zeiger auf die **Image**-Struktur der Grafik des Menü-Checkmarks. Das ist die Grafik, die in Auswahlmenüs erscheinen soll, wenn diese Auswahl (z.B. eine bestimmte Schriftart) aktiviert

ist. Steht hier eine Null, so wird die voreingestellte Grafik (ein Haken) benutzt.

***Title:** Der Zeiger auf den Textstring, der oben im Fenster als Titel erscheinen soll.

***Sceen:** Dieser Zeiger braucht Sie nur zu interessieren, wenn sie ein Custom-Screen eröffnet haben, und wollen, daß dieses Fenster dort geöffnet wird. In diesem Fall muß dieser Zeiger auf die entsprechende **Screen**-Struktur zeigen.

***Bitmap:** Wenn Sie für dieses Fenster eine eigene BitMap verwenden wollen (siehe auch **SUPERBITMAP**-Flag), so muß hier der Zeiger auf Ihre **BitMap**-Struktur stehen.

MinWidth, MinHeigth, MaxWidth, MaxHeight: Diese Variablen brauchen Sie nur zu interessieren, wenn Sie das Sizing-Gadget zum Verstellen der Fenstergröße mit der Maus, aktiviert haben. Soll es dem Benutzer also erlaubt sein, die Größe dieses Fensters zu verädern, so können Sie hier die Minimum-, bzw Maximumwerte eingeben, über die hinaus das Fenster nicht vergrößert oder -kleinert werden darf. Es handelt sich dabei wieder um relative x- und y-Koordinaten zur linken, oberen Ecke des Fensters.

Type: Hier steht, um welchen Fenster-Typ es sich handelt. Näheres darüber finden Sie in der **Screen**-Struktur.

Preferences

Die Datenstruktur **Preferences** beinhaltet sämtliche Einstellungen, die Sie von der Workbench aus eingestellt haben, oder die noch voreingestellt sind. Sie gehört zu den längsten Datenstrukturen, so daß wir sie hier nicht auflisten, sondern lediglich ein paar interessante Werte herauspicken und beschreiben. Dabei gibt die Zahl hinter der Variablen den Offset an, also ihre Position in der Struktur.

PointerMatrix USHORT, 28: Die Grafik des Default-Mauszeigers, zerlegt in 36 Wörter des Typen **USHORT**.

XOffset, YOffset BYTE, 100: Die x- und y-Position (von der linken oberen Ecke) des Aktivierungspunktes des Default-Mauszeigers, also z.B. die Position der Pfeilspitze.

color17, color18, color19 USHORT, 102: Die drei Farben des Default-Mauszeigers.

color0, color1, color2, color3 USHORT, 110: Die vier Farben des Workbench-Screens.

ViewXOffset, ViewYOffset BYTE, 118: Die Position des Workbench-Screen relativ zum Viewport.

ViewInitX, ViewInitY WORD,120: Die Initialisierungswerte des Offsets zum Viewport.

PrintImage USHORT, 168: Hier wird über das erste Bit angegeben, ob Grafiken negativ oder positiv zum Drucker ausgegeben werden. Ist das Bit gesetzt, so heißt das, daß der Grafikausdruck "negativ" erfolgt.

PrintAspect USHORT, 170: Ist hier das erste Bit gesetzt, so werden Grafiken vertikal, also um 90 Grad verdreht auf dem Drucker ausgegeben. Bei gelöschtem Bit erfolgt ein "normaler", horizontaler Ausdruck.

PrintThreshold WORD, 174: Haben Sie in **PrintShade** die Ausgabe in Graustufen ausgewählt, so können Sie hier die Graustufen den Farben zuordnen.

Screen

Dies ist bereits eine etwas längere Struktur, so daß wir sie hier zwar vollständig auflisten, jedoch uns in der Beschreibung auf das wichtigste Beschränken.

```
{
  struct Screen *NextScreen;
  struct Window *FirstWindow;
  SHORT  LeftEdge, TopEdge, Width, Height, MouseY,
  USHORT Flags;                                MouseX;
  UBYTE  *Title, *DefaultTitle;
  BYTE   BarHeight, BarVBorder, BarHBorder, MenuHBord
  BYTE   WBorTop, WBorLeft, WBorRight, WBorBottom;er;
  struct TextAttr *Font;J
  struct RastPort RastPort;
  struct BitMap BitMap;
  struct Layer_Info LayerInfo;
  struct Gadget *FirstGadget
  UBYTE  DetailPen, BlockPen;
  USHORT SaveColor0;
  struct Layer *BarLayer;
  UBYTE  *ExtData, *UserData;
};
```

***NextScreen:** Dies ist ein Zeiger auf den nächsten Screen in der Intuition-Screenliste. Ist er gleich **NULL**, so ist es der letzte, bzw. der einzige Screen.

***FirstWindow:** Hierbei handelt es sich um einen Zeiger auf das erste von mehreren zu dem Screen gehörenden Fenstern. Ist dieser Wert gleich **NULL**, so wurden keine Fenster definiert.

LeftEdge, TopEdge, Width, Height: Diese Werte geben die Dimensionen des Screens an und werden wie in der Struktur **NewScreen** benutzt.

MouseY, MouseX: Die Position des Mauszeigers auf diesem Screen.

Flags:

SCREENTYPE	0x000F	Wird von der Intution zur Erken-
WBENCHSCREEN	0x0001	nung des Screens verwendet (Work-
CUSTOMSCREEN	0x000F	bench- oder Custom-Screen).
SHOWTITLE	0x0010	Anzeigen der Menüleiste.
BEEPING	0x0020	Ist gesetzt, wenn der Screen gerade blinkt.
CUSTOMBITMAP	0x0040	Dieses Bit ist gesetzt, wenn Sie eine eigene BitMap benutzen.
SCREENBEHIND	0x0080	Der Screen wird im Hintergrund geöffnet.
SCREENQUIET	0x0100	Es ist keine Menüleiste auf diesem Screen vorhanden.

***Title, *DefaultTitle:** Zeiger auf den Titeltext des Screens und den Titeltext für Screens mit Fenstern.

BarHeight, BarVBorder, BarHborder, MenuVBorder, MenuHBorder: Die folgenden Variablen sind Größenangaben zu den Menüleisten und Rändern des Screens und allen

auf ihr erscheinenden Fenstern.Zunächst also die Höhe der Menüleiste dieses Screens in Pixels, sowie die vertikale und horizontale Breite des Menüleistenrandes.

WBorTop, WBorLeft, WBorRight, WBorBottom: Die Breite der Fensterränder. Getrennt nach oberem, linkem, rechtem und unterem Rand.

***Font:** Dies ist ein Zeiger auf den Zeichensatz, der innerhalb dieses Screens als der aktuelle benutzt werden soll.

Window

Eine solche Datenstruktur gehört zu jedem geöffneten Fenster. Sie ist relativ umfangreich und wir haben diese hier mehr der Vollständikeit halber aufgeführt. Bei der Erklärung beschränken wir uns wieder auf das Wesentliche. Ansonsten finden Sie die Grundlegenden Variablen und Flags auch in der Stuktur **NewWindow.**

```
struct Window
{
  struct Window *NextWindow;
  SHORT  LeftEdge, TopEdge, Width, Height, MouseY,
                                            MouseX;
  SHORT  MinWidth, MinHeight, MaxWidth, MaxHeight;
  ULONG  Flags;
  struct Menu *MenuStrip;
  UBYTE  *Title;
  struct Requester *FirstRequest;
  struct Requester *DMRequest;
```

```
  SHORT  ReqCount;
  struct Screen *WScreen;
  struct Rastport *RPort;
  BYTE   BorderLeft, BorderTop, BorderRight, Border-
                                             Bottom;
  struct RastPort *BorderRPort;
  struct Gadget *FirstGadget;
  struct Window *Parent, *Descendant;
  USHORT *Pointer;
  BYTE   PtrHeight, PtrWidth, XOffset, YOffset;
  ULONG  IDCMPFlags;
  struct MsgPort *UserPort, *WindowPort;
  struct IntuiMessage *MessageKey;
  UBYTE  DetailPen, BlockPen;
  struct Image *CheckMark;
  UBYTE  *ScreenTitle;
  SHORT  GZZMouseX, GZZMouseY, GZZWidth, GZZHeight;
  UBYTE  *ExtData;
  BYTE   *UserData;
  struct Layer *WLayer;
};
```

Flags: siehe ***NewWindow***.

***MenuStrip**: Dies ist ein Zeiger auf die Liste der Menüs. Diese werden in verketteten **Menu**- und **MenuItem**-Datenstrukturen gespeichert.

***Screen**: Zeiger auf den Screen, zu dem dieses Fenster gehört.

***Port**: Dies ist ein Zeiger auf dem zu diesem Fenster gehörigen Rastport.

***RPort**:Dies ist ein Zeiger auf den zu diesem Fenster, einschließlich des Fensterrandes, gehörigen Rastport.

***FirstGadget:** Ein Zeiger auf die Liste der zu diesem Fenster definierten Gadgets, aber ohne die Systemgadgets der Titelleiste. Diese werden über die IDCMP-Flags aktiviert.

***Pointer:** Ein Zeiger auf die Grafikdaten des Mauszei gers dieses Fensters. Dieser ersetzt den voreingestellten, wenn das zugehörige Fenster angeklickt wird und bleibt nur solange bestehen, wie das Fenster aktiviert ist.

PtrHeight, PtrHeight: Die Höhe und Breite der Grafik des Mauszeigers in Pixels, wobei die Breite kleiner oder gleich 16 sein muß.

XOffset,YOffset: Der Offset des Mauszeigers, also die x- und y-Koordinaten des Aktivierungspunktes der Mauszeigergrafik von der linken oberen Ecke aus. (Im allgemeinen wird es sich dabei um die Pfeilspitze oder das Zentrum eines Fadenkreuzes handeln.)

IDCMPFlags: Im allgemeinen gilt hier: ist ein Bit gesetzt, so erhalten Sie von der Intuition eine Message, falls der entsprechende Fall eintritt. Daher haben wir hier auch nicht alle Flags beschrieben. Für weitere Informationen vergleichen Sie dazu bitte die IDCMP-Flags der **IntuiMessage**-Struktur (weiter oben).

SIZEVERIFY 0x00000001

Sie bekommen eine Nachricht, wenn der Benutzer versucht, die Größe des Fensters zu verändern. Damit die Größe tatsächlich verändert werden kann, müssen Sie die Nachricht erst mal beantworten.

NEWSWIZE 0x00000002

Es wird eine IDCMP-Message gesendet, sobald sich die Größe des Fensters verändert.

REFRESHWINDOW 0x00000004

Bei **SIMPLE_REFRESH** und **SMART_REFRESH** Fenstern wird eine Message gesendet, wenn das Fenster "Refreshed" werden soll.

MOUSEBUTTONS 0x00000008

Sie werden benachrichtigt, wenn die Mausknöpfe betätigt werden. Ausnahme: die Intuition erkennt das anklicken von Gadgets selbstständig und meldet dies nicht. Das Drücken der rechten Maustaste wird nur dann gemeldet, wenn zusätzlich das **RMBTRAP-WindowFlag** gesetzt ist.

MOUSEMOVE 0x00000010

Mausbewgungen werden gemeldet, wenn auch das Flag **REPORTMOUSE** gesetzt ist.

GADGETDOWN 0x00000020

Es wird das Drücken der Maustaste über einem Gadget gemeldet.

GADGETUP 0x00000040

Es wird das Loslassen der Maustaste über einem Gadget gemeldet.

REQSET 0x00000080

Es wird Ihnen mitgeteilt, wenn in dem Fenster ein Requester geöffnet wird.

REQVERIFY 0x00000800
Im gegensatz zu **REQSET** werden Sie bereits vor dem Öffnen eines Requesters informiert.

MENUVERIFY 0x00002000

Sie werden benachrichtigt, wenn der Benutzer versucht ein Menü zu aktivieren. Das Menü wird erst dann wirklich aktiviert, wenn Sie diese Nachricht beantworten.

VANILLAKEY 0x00200000

Das Drücken einer Taste wird gemeldet. Der Tastatur-Code wird umgewandelt in den entsprechenden ASCII-Code.

INTUITICKS 0x00400000

Ist das Fenster aktiv, so erhält Ihr Programm ungefähr zehnmal pro Sekunde einen timer-event.

***UserPort, *WindowPort:** Zeiger auf zwei **MsgPort**-Exec-Strukturen, die dem Fenster zur Kommunikation mit der Außenwelt dienen (z.B. für die Console-Device).

GZZMouseX, GZZMouseY: Handelt es sich bei diesem Fenster um ein GZZ-Fenster, so stehen hier die x- und y-Koordinaten des Mauszeigers, diesmal jedoch relativ zu der linken, oberen Ecke innerhalb des Fensters. Rahmen und Titelleiste stehen in diesem Modus ausserhalb des Koordinatennetzes.

GZZWidth, GZZHeight: Bei einem GZZ-Fenster stehen hier die Dimensionen des inneren Fensters, ohne Rand und Titelleiste.

***ExtData, *UserData:** Diesen Zeiger können Sie auf eigene Erweiterungen setzten, die bei diesem Fenster ebenfalls beachtet werden sollen.

Anhang B

==
Die Datenstrukturen der Graphics-Library
==

Dieser Teil des Anhangs enthält die wichtigsten Datenstrukturen der Graphics-Library in alphabetischer Reihenfolge. Bei der Beschreibung der einzelnen Parameter haben wir uns, gerade bei den längeren Strukturen, auf die wesentlichsten beschränkt.

AreaInfo

Diese Struktur wird durch **InitArea** angelegt und enthält im wesentlichen die Koordinaten der einzelnen Punkte, die von den Area-Befehlen betroffen werden.

```
struct AreaInfo
{
SHORT *VctrTbl, *VctrPtr;
BYTE *FlagTbl *FlagPtr;
SHORT Count, MaxCount, FirstX, FirstY;
};
```

***VstrTbl:** Zeiger auf die Tabelle mit den Koordinatenwerten der entsprechenden Punkte.

***Vctrptr:** Zeiger auf den nächsten freien Platz in dieser Tabelle.

***FlagTbl, *FlagPtr:** ???

Count: Anzahl der Punkte, die in der Tabelle zur Zeit gespeichert sind.

MaxCount: Maximale Anzahl an Punkten, die in der Tabelle Platz finden können.

FirstX, FirstY: Startpunkt für die **AreaMove**-Routine.



Diese Datenstruktur enthält die Informationen über die zu einem RastPort angelegte Bitmap.

```
struct BitMap
{
UWORD BytesPerRow, Rows;
UBYTE Flags, Depth;
UWORD pad;
PLANEPTR Planes[8];
}
```

BytesPerRow: Breite der Bitmap in Bytes.

Rows: Höhe der Bitmap in Punkten.

Flags: Vom System benutzte Marke.

Depth: Anzahl der Bitplanes, die zu dieser Bitmap gehören

pad: Füllbyte

Planes: Tabelle mit den Zeigern auf die einzelnen Bitplanes dieser Bitmap.

ColorMap

Die **ColorMap** ist die Tabelle der Farbegister. Sie ist einem **RastPort** zugeordnet und enthält für diesen die bis zu 32 Farbwerte.

```
struct ColorMap
{
UBYTE Flags, Type;
UWORD Count;
APTR ColorTable;
};
```

Flags: Vom System immer noch ungenutzt.

Type: Bis jetzt ist die einzige Definition für diesen Wert Null.

Count: Aktuelle Anzahl der maximal 32 Farbeinträge in die Farbtabelle.

ColorTable: Zeiger auf den Beginn der Farbtabelle. Ein Eintrag besteht aus einem **ULONG**-Wert, dessen ersten vier Bits den Blauanteil, die nächsten vier Bits den Grünanteil und die folgenden Bits den Rotanteil der zugewiesenen Farbe beschreiben.

RasInfo

Eine **RasInfo**-Struktur gehört zu jedem Rastport. Sie stellt die Verbindung zu dem eigentlichen Display-Speicherbereich her. Der Display-Speicherbereich ist der Bereich, der schließlich direkt auf den Bildschirm gebracht wird.

```
struct RasInfo
{
struct RasInfo *Next;
struct BitMap *BitMap;
SHORT RxOffset, RyOffset;
};
```

***Next:** Zeiger auf die nächste asInfo-Struktur, nur im Zusammenhang mit dem Dual-Playfield-Modus benutzt.

***BitMap:** Zeiger auf die zu diesem Viewport gehörige Bitmap-Struktur.

RxOffset, RyOffset: Relative Koordinaten der Bitmap zum Viewport.

RastPort

Diese Datenstruktur ist wohl für die Grafikausgabe die wichtigste, da über diese alle Grafikbefehle ausgeführt werden. In der **RastPort**-Struktur finden wir

die wesentlichsten Voreinstellungen, wie z.B. Zeichenmodus und -Farben. Da dies wieder eine der längeren Datenstrukturen ist, haben wir wieder nur die wesentlichen Felder beschrieben.

```
struct RastPort
{
struct Layer *Layer, BitMap *BitMap;
USHORT *AreaPtrn;
struct TmpRas *TmpRas, AreaInfo *AreaInfo, Gels-
Info *GelsInfo;
UBYTE Mask;
BYTE FgPen, BgPen, AOlPen, DrawMode, AreaPtSz,
linpatcnt, dummy;
USHORT Flags, LinePtrn;
SHORT cp_x, cp_y;
UBYTE minterms[8];
SHORT PenWidth, PenHeight;
struct TextFont *Font;
UBYTE AlgoStyle, TxFlags;
UWORD TxHeight, TxWidth, TxBaseline;
WORD TxSpacing;
APTR *RP_User;
ULONG longreserved[2];
#ifndef GFX_RASTPORT_1_2
UWORD wordreserved[7];
UBYTE reserved[8];
#endif};
```

***Layer, *Bitmap:** Hierbei handelt es sich wieder um Zeiger auf die Adressen der zu diesem Rastport gehörigen **Layer**- und **Bitmap**-Strukturen.

***AreaPtrn:** Dies ist der Zeiger auf das zu diesem Rastport definierte Flächenfüllmuster.

Mask: Dies ist eine 8-Bit Maske, deren einzelnen Bits festlegen, ob in den dazugehörenden Bitmapebenen momentan geschrieben und gezeichnet werden kann. Ist ein Bit gesetzt, so kann die entsprechende Bitplane beschrieben werden, sonst nicht.

FgPen, BgPen: In diesen Feldern stehen die Farbcodes für die momentan aktive Vordergrund- und Hintergrundfarbe.

AOlPen: Der Farbcode für den Areafill-Outline-Pen-, den zweiten wichtigen Zeichenstift.

Drawmode: Der in diesem Rastport verwendete Zeichenmodus.

$00 JAM1:	Einfarbiges Raster.
$01 JAM2:	Zweifarbiges Raster.
$02 COMPLEMENT:	XOR-B Verknüpfung.
$04 INVERSVID:	Inverse Darstellung (z.B. von Text).

AreaPtSz: Enthält die Höhe des Flächenfüllmusters.

Flags: Folgende Werte sind definiert:

1	**FRST_DOT**	Erster Punkt einer Linie auch zeichnen.
2	**ONE_DOT**	Linien im EinBPunktBModus zeichnen.
4	**DBUFFER**	Rastport ist doppelt gepuffert.
8	**AREAOUTLINE**	Wird verwendet beim Flächenfüllen.
32	**NOCROSSFILL**	Kollision beim Flächenfüllen.

LinePtrn: Zeiger auf das zu diesem Rastport definierte Linienfüllmuster.

cp_x, cp_y: Momentane x- und y-Koordinaten des Grafikcursors.

PenWidth, PenHeight: Breite und Höhe des Zeichenstiftes, der von den Systemgrafikroutinen verwendet wird.

Font: Zeiger auf den Zeichensatz, der diesem Rastport zugeordnet ist.

TxHeight, TxWidth: Höhe und Durchschnittsbreite des aktuellen Zeichensatzes.

TxBaseline: Relativer Abstandswert

TextAttr

Enthält die Textattribute, wie z.B. die Schriftart, die bei einem Font gerade gültig sind.

```
struct TextAttr
{
STRPTR ta_Name;
UWORD ta_YSize;
UBYTE ta_Style;
UBYTE ta_Flags;
};
```

ta_Name: Name der momentan verwendeten Font.

ta_YSize: Höhe dieses Zeichensatztes in Pixel.

ta_Style: Momentan verwendete Schriftart, abzulesen aus folgenden gesetzten Bits:

0	**FS_NORMAL**	Textzeichen normal.
1	**FSB_UNDERLINED**	Textzeichen unterstrichen.
2	**FSB_BOLD**	Textzeichen fett.

```
4 FSB_ITALIC      Textzeichen kursiv.
8 FSB_EXTENDED    Textzeichen gedehnt.
```

ta_Flags: Folgende Bits sind definiert:

```
  1 FPB_ROMFONT        Font aus ROM geladen.
  2 FPB_DISKFONT       Font von Disk geladen.
  4 FPB_REVPATH        Von links nach rechts schreiben.
  8 FPB_TALLDOT        Font für HiRes-Modus.
 16 FPB_WIDEDOT        Font für LoRes/Interlace-Modus.
 32 FPB_PROPORTIONAL   Font mit Proportionalschrift.
 64 FPB_DESIGNED       Fontgröße nicht alg. gültig.
128 FPB_REMOVED        Font nicht aktiv.
```

TextFont

Eine solche Struktur gehört zu jedem Rastport und beinhaltet die Parameter zu dem aktuellen Font.

```
struct TextFont
{
struct Message tf_Message;
UWORD tf_YSize;
UBYTE tf_Style, tf_Flags;
UWORD tf_XSize, tf_Baseline, tf_BoldSmear,
UBYTE tf_LoChar, tf_HiChar;J                       tf_Accessors;
UWORD tf_Modulo;
APTR tf_CharLoc, tf_CharSpace, tf_CharKern;
};
```

tf_YSize: Die Höhe der Textzeichen des Fonts in Pixeln.

tf_Style, tf_Flags: Haben die gleiche Belegung wie in der TextAttr-Struktur.

tf_XSize: Die voreingestellte Breite der Textzeichen des Fonts in Pixeln.
tf_Baseline: Die Unterlänge der Textzeichen in einzelnen Zeilen.

tf_BoldSmear: Bei der Berechnung von "fetten" Zeichen verwendeter Schmierfaktor.

tf_Accessors: Anzahl der Routinen, Programme und Tasks, die auf diesen Font zugreifen.

tf_LoChar, tf_HiChar: Der kleinste, bzw. größte ASCII Wert, für den ein Textzeichen in diesem Font vorhanden ist.

tf_CharSpace: Bei Proportionalschrift ist dies ein Zeiger auf ein Feld von Ganzzahlen, die jeweils die Breite der einzelnen Textzeichen enthalten. Ist dieser Wert **NULL**, so gilt der in **tf_XSize** voreingestellte Wert.

TmpRas

Diese Struktur wird von den **Area**-Routinen gebraucht. Sie muß durch **InitTmpRas** initialisiert und einem Rastport zugeordnet werden.

```
struct TmpRas
{
BYTE *RasPtr;
LONG Size;
};
```

***RasPtr:** Zeiger auf einen Pufferspeicher.

Size: Größe des Pufferspeichers in Bytes. Sicherheitshalber sollte er die Größe einer Bitplane haben.

View

Beinhaltet die für die Aufteilung des Gesamt-Displays notwendigen Daten in die Viewports.

```
struct View
{
struct ViewPort *ViewPort, cprlist *LOFCprList,
cprlist *SHFCprList;
short DyOffset,DxOffset;
UWORD Modes;
};
```

***ViewPort:** Zeiger auf die zu dieser Struktur gehörenden ViewPort-Struktur.

***LOFCprList:** Zeiger auf die "Long-Frame-Copper-List".

***SHFCprList:**Zeiger auf die "Short-Frame-Copper-List".

DyOffset, DxOffset: x- und y- Koordinaten der Position der linken oberen Ecke dieses **ViewPorts** auf dem Monitor.

Modes: Folgende Werte sind definiert:

4 LACE	Interlace-Modus
64 PFBA	Erstes Playfield im Vordergrund

128	**EXTRA_HALFBRITE**	Extra-Halfbright-Modus
1024	**DUALPF**	Dual-Playfields-Modus
2048	**HAM**	Hold-And-Modify-Modus
8192	**VP_HIDE**	Ist Viewport im Hintergrund ?
32768	**HIRES**	High-Resolution-Modus

ViewPort

Diese Struktur enthält die Informationen der **Viewports**, über die der wesentliche Teil der Bildschirm-Ausgabe abläuft.

```
struct ViewPort
{
struct ViewPort *Next, ColorMap *ColorMap, CopList*
DspIns, CopList *SprIns, CopList *ClrIns, UCopList*
UCopIns;
SHORT DWidth,DHeight, DxOffset,DyOffset;
UWORD Modes;
UBYTE SpritePriorities, reserved;
struct RasInfo *RasInfo;
};
```

***Next**: Adreßzeiger auf den nächsten Viewport, falls noch einer vorhanden ist.

***ColorMap**: Zeiger auf die zu diesem Viewport gehörende **ColorMap**-Struktur.

DWidth, DHeight: Breite und Höhe dieses Viewports in Pixel.

DxOffset, DyOffset: Koordinatenpaar das die Position der linken oberen Ecke dieses Viewports auf dem Gesamtdisplay angibt.

Modes: Folgende Werte sind möglich:

4	**LACE**	Interlace-Modus
64	**PFBA**	Erstes Playfield vorn
128	**EXTRA_HALFBRITE**	Extra-Halfbright-Modus
1024	**DUALPF**	Dual-Playfields-Modus
2048	**HAM**	Hold-And-Modify-Modus
8192	**VP_HIDE**	Ist Viewport im Hintergrund ?
32768	**HIRES**	High-Resolution-Modus

***RasInfo**: Zeiger auf die zu diesem Viewport gehörende **RasInfo**-Struktur.

Anhang C

===
Die Systemfunktionen der Intuition-Library
===

Dieser Teil des Anhangs enthält die für die Grafikprogrammierung wichtigsten Systemfunktionen der Intuition-Library in alphabetischer Reihenfolge.

ClearPointer

Haben Sie sich zu einem Fenster einen eigenen Mauszeiger erstellt und diesen aktiviert, so können Sie ihn mit dieser Routine wieder desaktivieren. Es erscheint dann wieder der in den Preferences voreingestellte Mauszeiger.

```
ClearPointer(Window)
             A0
```

Window: Zeiger auf das Fenster, auf das sich diese Routine bezieht.

CloseScreen

Haben Sie einen eigenen Screen geöffnet, können Sie ihn mit dieser Routine wieder schließen, der entsprechende Speicherplatz wird wieder freigegeben. Zuvor alle Fenster durch **CloseWindow** schließen !

```
CloseScreen(Screen)
            A0
```

Screen: Adresse des Screens, der geschlossen werden kann.

CloseWindow

Mit dieser Routine können Sie ein Fenster schließen, der belegte Speicherplatz wird wieder freigegeben. Haben Sie einen IDCMP-Port geöffnet, so müssen Sie vorher sicherstellen, daß keine Nachricht mehr auf Antwort wartet.

```
CloseWindow(Window)
            A0
```

Window: Zeiger auf das zu schließende Fenster.

DrawBorder

Zeichnet die einer **Border**-Struktur entsprechenden Linien in dem angegebenen Rastport.

```
DrawBorder(RastPort, Border, LeftOffset, TopOffset)
           A0        A1      D0          D1
```

RastPort: Zeiger auf die **Rastport**-Struktur in der die Linien gezeichnet werden sollen.

Border: Zeiger auf die **Border**-Struktur in der die Informationen über die zu zeichnenden Linien stehen.

LeftOffset, TopOffset: Relative x- und y-Koordinaten, die zu jedem der Werte aus der **Border**-Struktur addiert werden.

DrawImage

Zeichnet ein Image an eine beliebige Stelle in einen Rastport.

```
DrawImage(RastPort, Image, LeftOffset, TopOffset)
          A0        A1     D0          D1
```

RastPort: Zeiger auf die **Rastport**-Struktur in der das Image gezeichnet werden soll.

Image: Zeiger auf die **Image**-Struktur in der die Informationen über das zu zeichnende Image stehen.

LeftOffset, TopOffset: Relative x- und y-Koordinaten, die zu dem Image addiert werden.

MoveScreen

Bewegt einen Screen relativ zu seiner jetzigen Position nach unten oder nach oben.

```
MoveScreen(Screen, DeltaX, DeltaY)
           A0      D0      D1
```

Screen: Zeiger auf die **Screen**-Struktur von dem Screen, den Sie verschieben wollen.

DeltaX, DeltaY: Setzen Sie **deltax** auf Null, da eine Verschiebung in der x-Achsen-Richtung noch nicht implementiert ist. Der Wert **deltay** bestimmt die Anzahl der Pixel, um die der Screen in vertikaler Richtung verschoben werden soll. Dabei bewirken negative Werte eine Verschiebung nach oben, positive nach unten.

MoveWindow

Diese Routine bewirkt durch die Intuition eine Verschiebung eines Fensters um angegebene Delta-Werte.

```
MoveWindow(Window, DeltaX, DeltaY)
           A0      D0      D1
```

Window: Adresse der **Window**-Struktur des Fensters, das verschoben werden soll.

DeltaX, DeltaY: Anzahl der Pixel, um die das Fenster verschoben werden soll. Positive Werte bewirken eine Verschiebung nach unten (bzw. nach rechts), negative Werte in die entgegengesetzte Richtung.

Wichtig(!): Bevor Sie diese Routine aufrufen, prüfen Sie unbedingt, ob das Fenster nach dem Verschieben

noch komplett im Gültigkeitsbereich (innerhalb des Screens) liegt, da Ihnen sonst ein unvermeidlicher Absturz bevorsteht.

OpenScreen

Öffnet einen neuen Screen mit den in einer **NewScreen-Struktur** (siehe Anhang A) angegebenen Parametern und legt eine Screen-Struktur an. Die NewScreen-Struktur wird anschließend nicht mehr benötigt und kann aus dem Speicher gelöscht werden.

```
ScreenPointer = OpenScreen(NewScreen);
                               A0
```

NewScreen: Zeiger auf eine NewScreen-Struktur, die die für diesen Screen geltenden Parameter enthält.

Rückgabe: ScreenPointer wird **NULL**, wenn der Screen nicht geöffnet werden konnte, ansonsten enthält er die Adresse der **Screen**-Datenstruktur des geöffneten Screens.

OpenWindow

Öffnet ein Fenster mit den in einer **NewWindow**-Struktur (siehe Anhang A) angegebenen Parametern und legt eine **Window**-Struktur an. Die **NewWindow**-Struktur wird anschließend nicht mehr benötigt und kann aus dem Speicher gelöscht werden.

```
WindowPointer = OpenWindow(NewWindow);
                           A0
```

NewWindow: Zeiger auf eine **NewWindow**-Struktur, die die für dieses Fenster geltenden Parameter enthält.

Rückgabe: WindowPointer wird **NULL**, wenn das Fenster nicht geöffnet werden konnte, ansonsten enthält er die Adresse des geöffneten Fensters.

PrintIText

Ein durch die **IntuiText**-Struktur vorgegebener Text wird in dem angegebenen R astport aneiner durch die x-und y-Koordinaten bestimmte Stelle ausgegeben.

```
PrintIText(RastPort, IText, LeftEdge, TopEdge)
           A0        A1     D0        D1
```

RastPort: Zeiger auf den Rastport in dem der Text ausgegeben werden soll.

IText: Zeiger auf die **IntuiText**-Struktur mit dem auszugebenden Text und den dazugehörigen Parametern über die Schriftart.

LeftEdge, TopEdge: Position des Textes innerhalb des Rastports, relativ zur linken, oberen Ecke.

ScreenToBack

Setzt einen beliebigen Screen in den Hintergrund.

```
ScreenToBack(Screen)
             A0
```

Screen: Zeiger auf den in den Hintergrund zu setzenden Screen.

ScreenToFront

Setzt einen beliebigen Screen in den Vordergrund.

```
ScreenToFront(Screen)
              A0
```

Screen: Zeiger auf den in den Vordergrund zu setzenden Screen.

SetPointer

Ordnet einem Fenster einen eigenen Mauszeiger zu, der immer dann erscheint, wenn das Fenster aktiv ist.

```
SetPointer(Window,Pointer,Height,Width,XOff,YOff);
           A0     A1      D0     D1    D2   D3
```

Window: Zeiger auf das Fenster für das ein eigener Zeiger definiert werden soll.

Pointer: Zeiger auf die Grafik-Daten des Mauszeigers.

Height, Width: Höhe und Breite des Zeigers, wobei die Breite kleiner oder gleich 16 sein muß.

XOff, YOff: Offset-Werte für den "Hot-Spot" relativ zur linken oberen Ecke. Handelt es sich bei dem Zeiger um einen Pfeil, der seine Spitze in der linken oberen Ecke hat, dann setzen Sie diese Werte auf Null. Ist dagegen die Pfeilspitze oben rechts, so setzen Sie XOff auf 15 (vorausgesetzt der Zeiger ist 16 Punkte breit).

ShowTitle

Mit Hilfe dieser Routine können Sie die Screen-Titelleiste bei Überlagerung mit einem BackDrop-Fenster in den Vorder- oder Hintergrund bringen.

```
Showitle(Screen, ShowIt);
         A0      D0
```

Screen: Zeiger auf den Screen mit der zu beeinflussenden Screen-Titelleiste.

ShowIt: Setzen Sie diesen Wert auf TRUE um die Titelleiste in den Vordergrund und auf FALSE um sie in den Hintergrund zu bringen.

SizeWindow

Diese Routine fordert Intuition auf, ein Fenster um die angegebenen Delta-Werte zu vergrößern, bzw. zu verkleinern. Intuition führt dann diese Veränderung ohne vorherige Kontrolle der übergebenen Werte durch.

```
SizwWindow(Window, DeltaX, DeltaY);
           A0      D0      D1
```

Window: Zeiger auf das Fenster, dessen Größe verändert werden soll.

DeltaX, DeltaY: Anzahl an Punkten, um die die Größe des Fensters in x- und y-Richtung verändert werden soll. Dabei bedeutet ein negatives Vorzeichen verkleinern, positives vergrößern.

Bemerkung: Um einen Systemabsturz mit den Worten der Orginal-Beschreibung (siehe Literaturverzeichnis) zu umschreiben: "Diese Routine macht keine Fehlerkontrolle! Beschreiben Ihre Delta-Werte irgendeine entfernte Ecke des Universums, so wird Intuition versuchen, das Fenster bis dorthin zu vergrößern. Aufgrund der Dehnungen die im Raum-Zeit Kontinuum,vorhergesagt durch die Spezielle Relativität, entstehen können, ist das Ergebnis dieses Versuches im allgemeinen nicht wünschenswert."

ViewPortAddress

Bestimmt zu einem beliebigen Fenster die Adresse des zugehörigen Viewports. Diese Adresse brauchen Sie für die meisten Grafik- und Text-Routinen.

```
ViewPort = ViewPortAddress(Window);
                           A0
```

Window: Zeiger auf das Fenster, von dem Sie die Adresse des Viewports haben wollen.

WindowLimits

Mit dieser Routine können Sie zu einem Fenster neue Maximum- und Minimum-Dimensionen setzen. Mit dem Sizing-Gadget kann dann dieses Fenster nur innerhalb dieser Werte vergrößert oder verkleinert werden.

```
boole = WindowLimits(Window,MinX,MinY,MaxX,MaxY);
                     A0     D0   D1   D2   D3
```

Window: Zeiger auf das Fenster, dessen maximale und minimale Dimension Sie festlegen wollen.

MinX, MinY: Minimale x- und y- Werte, die beim Verkleinern des Fensters nicht mehr über- bzw. unterschritten werden können.

MaxX, MaxY: Maximale x- und y- Werte, die beim Vergrößern des Fensters nicht mehr über- bzw. unterschritten werden können.

WindowToBack

Veranlaßt Intuition ein ausgewähltes Fenster in den Hintergrund zu setzen.

```
WindowToBack(Window)
             A0
```

Window: Zeiger auf das in den Hintergrund zu setzende Fenster.

WindowToFront

Veranlaßt Intuition ein ausgewähltes Fenster in den Vordergrund zu setzen.

```
WindowToFront(Window)
              A0
```

Window: Zeiger auf das in den Vordergrund zu setzende Fenster.

Anhang D

===
Die Systemfunktionen der Graphics-Library
===

Dieser Teil des Anhangs enthält die fÜr die Grafikprogrammierung wichtigsten Systemfunktionen der Graphics-Library in alphabetischer Reihenfolge.

AddFont

Fügt einen Font in die System-Font-Liste ein, damit er im Speicher zur schnelleren Verfügung steht.

```
AddFont(TextFont), GraphicsLib
        A1         A6
```

TextFont: Ein Zeiger auf eine **TextFont**-Struktur des einzufügenden Fonts.

AllocRaster

Ruft die notwendigen **Allocate**-Routinen auf, um den Speicherplatz für eine Bitplane zu belegen.

```
AllocRaster( width, height)
             D0     D1
```

width, height: Breite und Höhe der Bitplane, für die dieser Speicherplatz reserviert werden soll.

Rückgabe: Ein Zeiger auf den Anfang des reservierten Speicherplatzes oder 0 falls die Belegung nicht erfolgreich war.

AreaDraw

Ein weitere Punkt des auszufüllenden Polygons wird in die Liste der Polygon-Eckpunkte eingefügt.

```
error = (int) AreaDraw( RastPort,  x,  y)
                        A1         D0  D1
```

RastPort: Zeiger auf die **RastPort**-Struktur, in der das Polygon gezeichnet werden soll.

x, y: Koordinaten des Polygon-Eckpunktes.

Rückgabe: Bei erfolgreicher Durchführung 0 sonst -1.

AreaEnd

Zeichnet ein ausgefülltes Polygon, dessen Eckpunkte durch **AreaDraw** eingegeben wurden.

```
error = AreaEnd(RastPort)
                A1
```

RastPort: Zeiger auf die **RastPort**-Struktur, in der das Polygon gezeichnet werden soll.

Rückgabe: Bei erfolgreicher Durchführung 0 sonst -1.

AreaMove

Setzt den Startpunkt eines zu füllenden Polygons. Sollte das letzte Polygon noch nicht abgeschlossen worden sein, so wird dies von dieser Routine automatisch erledigt.

```
error = AreaMove( RastPort,  x,  y)
                  A1         D0  D1
```

RastPort: Zeiger auf die **RastPort**-Struktur, in der das Polygon gezeichnet werden soll.

x, y: Koordinaten des Polygon-Startpunktes.

Rückgabe: Bei erfolgreicher Durchführung 0 sonst -1.

AskFont

Erzeugt eine TextAttr-Struktur, die die Parameter des aktuellen Zeichensatzes enthalten.

```
Askfont(RastPort, TextAttr)
        A1        A0
```

RastPort: Zeiger auf den **RastPort**, dessen Font-Attribute in der **TextAttr**-Struktur abgelegt werden sollen.

TextAttr: Zeiger auf die **TextAttr**-Struktur, in der die Informationen abgelegt werden.

AskSoftStyle

Mit Hilfe dieser Routine können Sie erfahren, welche Schriftarten des aktuellen Fonts eines Rastports Ihnen noch zur Verfügung stehen.

```
bits = AskSoftStyle(RastPort)
                    A1
```

RastPort: Zeiger auf den Rastport, von dessen Font Sie wissen wollen, welche Schriftarten Ihnen zur Verfügung stehen.

Rückgabe: Liefert die Font-Style-Flags zurück, die noch durch **SetSoftStyle** gesetzt werden können.

BltBitMap

Mit dieser Routine können Sie den Blitter veranlassen, ein Rechteck aus einer Bitmap in eine andere zu kopieren, innerhalb einer Bitmap zu verschieben oder mit einem anderen Rechteck zu verknüpfen.

```
planes = BltBitMap(Srcx, Srcy,Destx,Desty,width,
D0                 D0    D1   D2    D3    D4

  height, Minterm, Mask, ScrBitMap, DestBitMap,Tempa)
  D5      D6       D7    A0         A1         A2
```

ScrBitMap: Zeiger auf die Source-Bitmap (Quell-Bitmap), also diejenige von der Sie ein Rechteck kopieren wollen.

Srcx, Srcy: Koordinaten der linken oberen Ecke des zu kopierenden Rechtecks innerhalb der Quell-Bitmap.

DestBitMap: Zeiger auf die Destination-Bitmap (Ziel-Bitmap),also diejenige in die Sie ein Rechteck hinein kopieren wollen. Diese kann mit der Source-Bitmap identisch sein.

Destx, Desty: Koordinaten innerhalb der Ziel-Bitmap, an der dann die linke obere Ecke des kopierten Rechteckes liegt.

width, heigth: Breite und Höhe des kopierten Rechtecks in Pixel.

Minterm: Die logische Verknüpfung, die beim Kopieren des Rechteckes in das Zielrechteck vorgenommen wird. Bezeichnen wir das Quellrechteck mit Q und das Zielrechteck mit Z, so bewirken die einzelnen Bits folgende Verknüpfungen:

0x10 Q AND Z
0x20 Q AND NOT(Z)
0x40 NOT(Q) AND Z
0x80 NOT(Q) AND NOT(Z)

So bewirkt das Setzen von mehreren dieser Bits zum Beispiel:

0x30 **Invertieren von Q.**
0x50 **Invertieren von Z.**
0xC0 **Kopieren ohne Werte zu verändern.**

Mask: Hier können Sie durch Setzen der einzelnen Bits festlegen, welche einzelnen Bitplanes einer Bitmap kopiert werden sollen. Setzen sie zum Beispiel Bit 0 auf 1, so wird die Bitplane 1 kopiert.

TempA: Zeiger auf einen Zwischenspeicher, der nur benutzt wird, wenn sich das Quellrechteck mit dem Zielrechteck überschneidet.

BltBitMapRastPort

Mit dieser Routine können Sie den Blitter veranlassen ein Rechteck aus einer Bitmap in einen **RastPort** zu kopieren. Die Parameter sind im wesentlichen die gleichen wie bei **BltBitMap**. Daher haben wir hier auch nur den von **BltBitMap** verschiedenen Parameter angegeben.

```
boole= BltBitMapRastPort(Srcx,Srcy,Destx,Desty,width,
                         D0   D1   D2    D3    D4
           height,Minterm,ScrBitMap,DestRast)
           D5     D6      A0        A1
```

DestRast: Zeiger auf den Destination-RastPort (Ziel-RastPort).

Destx, Desty: Relative Koordinaten innerhalb des Ziel-RastPortes, an der dann die linke obere Ecke des kopierten Rechteckes liegt.

BltClear

Löscht einen angegebenen Speicherbereich (= belegen mit 0) im Chip-RAM.

```
BltClear(memblock, bytecount, flags)
         A1        D0         D1
```

bytecount: Anzahl der zu löschenden Bytes in Abhängigkeit von flags.

flags: Ist Bit 0 gesetzt, so wartet das Programm, bis der Blitter den Löschvorgang beendet hat, ansonsten läuft es während des Löschens weiter. Ist Bit 1 gelöscht, so werden soviel Bytes gelöscht wie in bytecount angegeben, ist es gesetzt, so wird ein Rechteck gelöscht, das durch die oberen und unteren 16 Bits von bytecount gegeben ist. Dabei entsprechen die unteren 16 Bit der Anzahl der zu löschenden Bytes per Zeile und die oberen 16 Bit der Anzahl der zu löschenden Zeilen.

BltMaskBitMapRastPort

Diese Funktion bewirkt im Wesentlichen das gleiche wie **BltBitMap.** Sie haben jedoch zusätzlich die Möglichkeit beim kopieren eine Bitmap als "Maske" zu benutzen, die wie eine Schablone wirkt und nur da durchlässig ist, wo die Bits gesetzt sind. Da auch hier die Parameter größtenteils mit denen von **BltBit-**

Map und **BltBitMapRastPort** übereinstimmen, haben wir hier nur den hinzugekommenen dokumentiert.

```
boole = BltMaskBitMapRastPort
        (ScrBitMap, Srcx, Srcy, DestRast
        A0          D0    D1    A1
       Destx, Desty, width, height, Minterm, BltMask)
       D2     D3     D4     D5      D6       A2
```

BltMask: Zeiger auf die Bitplane, die als Maske dient.

BltPattern

Füllt ein Rechteck, in Abhängigkeit von einer Maske (wie bei **BltMaskBitMapRastPort**), in einen Rastport mit den aktuellen Parametern (Farbe, Füllmuster etc.)

```
BltPattern( RastPort, Mask, x1, y1, x2, y2, Bytes )
            A1        A0    D0  D1  D2  D3  D4
```

RastPort: Zeiger auf den Rastport, in dem ein Rechteck gezeichnet werden soll.

Mask: Zeiger auf eine BitPlane, die mindestens so groß sein muß wie Bytes und die beim Füllen als Schablone dient.

x1, y1, x2, y2: Koordinaten der linken oberen und der unteren rechten Ecke des Rechtecks.

Bytes: Breite des Rechtecks in Bytes. Also in 8er Schritten aufgerundet, da 1 Byte = 8 Bit.

ClearEOL

Löscht eine Zeile ab der aktuellen Cursor-Position bis zum Ende der Zeile (End Of Line). Die Breite der gelöschten Zeile hängt von der aktuellen Text-Höhe ab.

```
ClearEOL(RastPort), GraphicsLib
         A1         A6
```

RastPort: Zeiger auf den RastPort, in dem die Zeile gelöscht werden soll.

DisownBlitter

Freigeben des Blitters für andere Tasks, nachdem er durch OwnBlitter für einen Task reserviert wurde.

```
DisownBlitter()
```



Zeichnet eine Linie zwischen der aktuellen Cursor-Position und den angegebenen Koordinaten.

```
Draw( RastPort,  x,  y)
      A1         D0  D1
```

RastPort: Zeiger auf den **RastPort**, in der die Linie gezeichnet werden soll.

x, y: Koordinaten der neuen aktuellen Cusor-Position, die mit der alten durch eine Linie verbunden wird.



Zeichnet im angegebenen RastPort eine Ellipse.

```
DrawEllipse( RastPort, xm, ym, xr, yr)
             A1        D0  D1  D2  D3
```

RastPort: Zeiger auf den **RastPort**, in der die Ellipse gezeichnet werden soll.

xm, ym: Koordinaten des Mittelpunktes der Ellipse.
xr, yr: x- und y-Radius der Ellipse.



Füllen einer zusammenhängenden Fläche.

```
Flood( RastPort, mode,  x,  y)
       A1        D2     D0  D1
```

RastPort: Zeiger auf den **RastPort**, in der sich die zu füllende Fläche befindet.

mode: Füllmodus

x, y: x- und y-Koordinate des Punktes, ab dem mit dem Füllen begonnen wird.

FreeColorMap

Löschen der **ColorMap**-Struktur durch Freigegeben des von ihr belegten Speichers.

```
FreeColorMap( colormap )
              A0
```

colormap: Zeiger auf die **ColorMap**-Struktur, die freigegeben werden kann.

FreeRaster

Freigeben eines Speicherbereiches, der von einer Bitmap durch die **AllocRaster**-Routine belegt wird.

```
FreeRaster(  p, width, height)
            A0  D0     D1
```

p: Zeiger auf den Speicherbereich, der durch die Bitplane belegt wird.

width, height: Breite und Höhe der Bitplane, deren Speicherplatz wieder freigegeben werden kann.

Wichtig(!): Benutzen Sie für **width** und **height** die gleichen Werte die Sie auch bei **AllocRaster** verwendet haben! Sie laufen sonst Gefahr eine "Guru Meditation" zu erhalten.

GetColorMap

Diese Routine legt eine **ColorMap**-Struktur an.

```
cm = GetColorMap( entries )
D0                D0
```

entries: Anzahl der Farben, die in die Liste einge- tragen werden sollen.

Rückgabe: Zeiger auf die initialisierte **ColorMap**-Struktur.

GetRGB4

Routine zum Auslesen der einzelnen Farbwerte aus den Farbregistern.

```
value = GetRGB4( colormap, entry )
D0               A0        D0
```

colormap: Zeiger auf die **ColorMap**-Struktur, aus der Sie die Farbwerte lesen wollen.

entry: Nummer des Farbregisters, von dem Sie die Farbwerte wissen wollen.

Rückgabe: Sie erhalten -1, wenn das Farbregister mit keiner gültigen Farbe belegt ist. Ansonsten ist value

ein Wort dessen ersten vier Bits (0-3) den Blauanteil die nächsten vier Bits (4-7), den Grünanteil und die folgenden vier Bits (8-11), den Rotanteil der Farbe beinhalten.

InitArea

Bevor Sie die **Area**-Befehle benutzen können, müssen Sie durch diese Routine eine **AreaInfo**-Struktur initialisieren.

```
InitArea(AreaInfo, Buffer, Count)
         A0        A1      D0
```

AreaInfo: Zeiger auf die zu initialisierende Area-Info-Struktur.

Buffer: Zeiger auf einen freien Speicherbereich, der den Area-Befehlen als Zwischenspeicher dient.

Count: Maximale Anzahl der Punkte die im Zwischenspeicher platz finden können. Für jeden Punkt werden dann 5 Bytes Speicher reserviert.

InitBitMap

Diese Routine dient dazu, eine **BitMap**-Struktur zu initialisieren.

```
InitBitMap( BitMap, depth, width, height)
            A0      D1     D2     D3
```

BitMap: Zeiger auf die zu initialisierende BitMap.

depth: Anzahl der Bitplanes, die diese Bitmap hat.

width, height: Breite und Höhe der Bitmap in Pixel.

InitRastPort

Initialisieren einer **RastPort**-Struktur mit den Standardwerten (Modus = **JAM2**; Mask, FgPen, AOLPen und LinePtrn = -1; restliche Werte = 0)

```
InitRastPort(RastPort)
             A1
```

RastPort: Zeiger auf den zu initialsierenden **Rast-Port.**

InitTmpRas

Initial.isieren einer **TmpRas**-Struktur, die einem **Rast Port** als Zwischenspeicher dient.

```
InitTmpRas(tmpras, buffer, size)
           A0      A1      D0
```

tmpras: Zeiger auf die zu initialisierende TnpRas-Struktur.

buffer: Zeiger auf einen freien Speicherbereich.

size: Größe von **buffer** in Bytes. Im allgemeinen sollten Sie den Zwischenspeicher so groß wie eine Bitplane dieses **RastPorts** wählen.

InitView

Diese Routine initialisiert die angegebene ViewPort Struktur durch Setzen der wichtigsten Werte.

```
InitView( view )
          A1
```

view: Zeiger auf die zu initialisierende **ViewPort**-Struktur.

InitVPort

Initialisiert eine **ViewPort**-Struktur mit den Standard Werten.

```
InitVPort(ViewPort)
          A0
```

ViewPort: Zeiger auf die zu initialisierende **ViewPort**-Struktur.

LoadRGB4

Kopiert eine Farbpalette mit den entsprechenden Einträgen in einen Viewport.

```
LoadRGB4(ViewPort, ColorMap, Count)
         A0        A1        D0
```

ViewPort: Zeiger auf den **ViewPort**, dessen Farbeinträge Sie ändern möchten.

ColorMap: Zeiger auf einen Array von **UWORD**-Variablen, die die Farbwerte der Farbregister enthalten.

Count: Anzahl der Farbregister die in dem Array **Color Map** gespeichert sind.

LoadView

Durch diese Routine werden die Copperlisten, die Sie durch **MakeVPort** berechnet haben, ausgeführt.

```
LoadView (View);
          A1
```

View: Adresse der **View**-Struktur, die die bereits berechneten Copperlisten enthält.

MakeVPort

Bereitet einen Viewport auf, indem seine Zwischen-Copperlisten berechnet werden.

```
MakeVPort( View, ViewPort);
           A0    A1
```

View: Zeiger auf die **View**-Struktur, dem dieser Viewport zugeordnet ist.

ViewPort: Zeiger auf die **ViewPort**-Struktur, deren Zwischen-Copperlisten berechnet werden sollen.

Move

Bewegt den grafischen Zeichenstift zu einer (x,y)-Position, (relativ zur linken oberen Ecke des Rastports) ohne dabei irgend etwas zu zeichnen.

```
Move( RastPort,  x,  y);
      A1        D0  D1
```

RastPort: Adresse der **RastPort**-Struktur, in der die Position des Zeichenstiftes neu gesetzt wird.

x, y: Koordinaten des Punktes, auf den der Zeichenstift gesetzt werden soll.

OpenFont

Öffnen des Fonts der System-Font-Liste, der am ehesten einer vorgegebenen **TextAttr**-Struktur entspricht.

```
font = OpenFont(TextAttr), GraphicsLib
D0              A0          A6
```

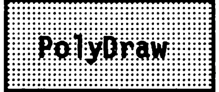
TextAttr: Zeiger auf die **TextAttr**-Struktur, zu der ein Font geladen werden soll.

Rückgabe: font ist 0, falls kein passender Font gefunden wurde, sonst enthält font den Zeiger auf die initialisierte **TextFont**-Struktur.

OwnBlitter

Reserviert den Blitter für Ihre eigenen Zwecke. Stellen Sie jedoch vorher durch **WaitBlit** sicher, daß der Blitter gerade nicht arbeitet.

```
OwnBlitter();
```



Zeichnet einen vorher definierten Linienzug (bzw. ein Polygon, falls der Startpunkt gleich dem EndPunkt ist) in den angegebenen **RastPort**.

```
PolyDraw( RastPort, count, array )
          A1        D0     A0
```

RastPort: Zeiger auf den **RastPort**, in dem der Linienzug gezeichnet werden soll.

count: Anzahl der vorher definierten Eckpunkte.

array: Zeiger auf den Speicherbereich, wo die Koordinaten der Eckpunkte stehen.

ReadPixel

Liest die Farbe eines Punktes eines Rastports.

```
pen = (int)ReadPixel( RastPort,  x,  y )
D0                    A1        D0  D1
```

RastPort: Zeiger auf den **RastPort**, aus dem ein Punkt "gelesen" werden soll.

x, y: Relative (zur linken oberen Ecke des **RastPorts**) Koordinaten des Punktes, von dem Sie wissen wollen, mit welcher Farbe er gesetzt wurde.

Rückgabe: Liefert -1 falls der Punkt nicht gelesen werden kann, ansonsten die Nummer des entsprechenden Farbregisters.

RectFill

Diese Routine zeichnet ein ausgefülltes Rechteck mit den aktuellen Werten (z.B.: ZeichenModus, Farbe, Füllmuster, etc.).

```
RectFill( RastPort, xmin, ymin, xmax, ymax)
          A1        D0    D1    D2    D3
```

RastPort: Zeiger auf den **RastPort**, in dem das Rechteck gezeichnet werden soll.

xmin, ymin: Koordinaten der linken oberen Ecke des zu zeichnenden Rechtecks.

xmax, ymax: Koordinaten der rechten unteren Ecke des zu zeichnenden Rechtecks.

RemFont

Entfernt einen Font aus der System-Font-Liste.

```
error = RemFont(TextFont),GraphicsLib
D0              A1        A6
```

TextFont: Adresse der **TextFont**-Struktur, die wieder freigegeben werden kann.

Rückgabe: Ist error = 0, so konnte der Font nicht entfernt werden.

ScrollRaster

Mit dieser Routine können Sie ein Rechteck in einem Rastport scrollen, also seinen Inhalt verschieben.

```
ScrollRaster(RastPort, dx, dy,Xmin,Ymin,Xmax,Ymax);
             A0        D0  D1  D2   D3   D4   D5
```

RastPort: Zeiger auf den **Rastport**, in dem Sie ein Rechteck scrollen wollen.

dx, dy: Anzahl der Punkte, um die der Inhalt des Rechtecks verschoben wird. Positive Werte bewirken eine Verschiebung nach links, negative nach rechts.

Xmin, Ymin: Koordinaten der linken oberen Ecke des Rechtecks.

Xmax, Ymax: Koordinaten der linken oberen Ecke des Rechtecks.

SetAPen

Setzen der neuen Vordergrundfarbe (Primary pen).

```
SetAPen( RastPort, pen)
         A1        D0
```

RastPort: Zeiger auf den **RastPort**, für den dieser Farbstift neu gesetzt werden soll.

pen: Nummer des Farbregisters, mit dessen Farbe dieser Farbstift belegt wird.

SetBPen

Setzen der neuen Hintergrundfarbe (Secondary pen).

```
SetBPen( RastPort, pen)
         A1        D0
```

RastPort: Zeiger auf den **RastPort**, für den dieser Farbstift neu gesetzt werden soll.

pen: Nummer des Farbregisters, mit dessen Farbe dieser Farbstift belegt wird.

SetDrMd

Setzt einen neuen Zeichenmodus.

```
SetDrMd( RastPort, Mode);
         A1        D0
```

RastPort: Addresse der **RastPort**-Struktur, deren Zeichenmodus Sie neu festlegen möchten.

Mode: Die folgenden vier Modi können auch beliebig miteinander verknüpft werden, wenn dies auch nicht immer sinnvoll ist.

JAM1	0	Gesetzte Bits mit APen zeichnen.
JAM2	1	Wie JAM1, aber gelöschte Bits mit BPen zeichnen.
COMPLEMENT	2	Punkte vorm zeichnen "XOR"en.
INVERSVID	4	Invertieren (z.B von Textzeichen)

SetFont

Ordnet dem angegebenen **RastPort** einen neuen Zeichensatz zu.

```
error = SetFont( RastPort, Font),GraphicsLib
D0               A1        A0    A6
```

RastPort: Zeiger auf den **RastPort**, für den der Font neu gesetzt werden soll.

Font: Zeiger auf die bereits durch **OpenFont** vorbereitete **TextFont**-Struktur.

SetOPen

Setzen der neuen Umrandungsfarbe (Outline pen).

```
SetOPen( RastPort, pen)
         A1        D0
```

RastPort: Zeiger auf den **RastPort**, für den dieser Farbstift neu gesetzt werden soll.

pen: Nummer des Farbregisters, mit dessen Farbe dieser Farbstift belegt wird.

SetRGB4

Mit dieser Routine können Sie die Farbregister eines Viewports mit neuen Farben belegen.

```
SetRGB4( ViewPort, Number, Red, Green, Blue);
         A0        D0      D1   D2     D3
```

ViewPort: Zeiger auf die **ViewPort**-Struktur.

Number: Nummer des Farbregisters (0-31).

Red, Green, Blue: Die neuen Farbanteile (0-15).

Text

Gibt an der aktuellen Position des Zeichenstiftes einen beliebigen Text aus.

```
error = Text(RastPort, String, Count)
D0             A1        A0      D0
```

RastPort: Zeiger auf den **RastPort**, in dem der Text ausgegeben wird.

String: Adresse der Zeichenkette, die Sie ausgegeben möchten.

Count: Anzahl der einzelnen Zeichen, die die Zeichenkette enthält.

Rückgabe: Bei erfolgreicher Durchführung 0 sonst -1.

TextLength

Mit dieser Routine können Sie die eigentliche Länge (in Punkten) eines Textes, unter Berücksichtigung des aktuellen Fonts, ermitteln.

```
length = TextLength(RastPort, String, Count);
D0                    A1        A0      D0-0:16
```

RastPort: Zeiger auf den RastPort, in dem der Text erscheinen soll.

String: Adresse der Zeichenkette, die Sie ausgegeben möchten.

Count: Anzahl der einzelnen Zeichen, die die Zeichenkette enthält.

VBeamPos

Gibt Ihnen die Zeilennummer an, in der sich der Elektronenstrahl, der das aktuelle Bild wiedergibt, befindet. Der Wert ist im allgemeinen recht ungenau, wir verwenden ihn lediglich, um eine Zufallszahl zu erhalten.

```
position = VBeamPos();
```

Rückgabe: Ein Wert zwischen 0 und 255.

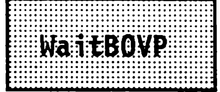
WaitBlit

Wartet bis der Blitter seine augenblickliche Kopierarbeit beendet hat, damit Sie die Routine **Own-Blitter** benutzen können.

```
WaitBlit();
```

Bemerkung: Diese Routine arbeitet leider nicht immer fehlerfrei, sondern verzichtet manchmal auf weiteres

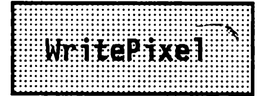
Warten, obwohl der Blitter noch einen Kopiervorgang zu erledigen hat.



Wartet bis der Elektronenstrahl, der das Bild eines Viewports zeichnet, an seinem unteren Rand angekommen ist.

```
WaitBOVP(ViewPort);
         A0
```

ViewPort: Zeiger auf den Viewport, auf dessen vollständige Darstellung Sie warten wollen.



Setzt einen Pixel an der Position x,y.

```
Writepixel( RastPort,  x,  y)
            A1         D0  D1
```

RastPort: Zeiger auf den **RastPort**, in dem ein Punkt gesetzt werden soll.

x, y: Relative (zur linken oberen Ecke des **RastPorts**) Koordinaten des zu setzenden Punktes.

Anhang E

===

Die Blitter-Hardware-Registerbeschreibung

===

In diesem Anhang finden Sie für die Hardwareprogrammierung des Blitters eine kurze Beschreibung seiner Register.

BLTAFWM (044)

(BLiTter: source A, First Word Mask)

BLTALWM (046)

(BLiTter: source A, Last Word Mask)

Die Bitmuster dieser beiden Register werden mit dem ersten, beziehungsweise letzten Wort jeder kopierten Daten-Zeile "geANDet". Somit ist es möglich den Anfang und das Ende einer zu kopierenden Daten-Zeile der Quelle A nicht nur Wortweise, sondern auch Bitweise festzulegen. Beim Füllen oder Lienien zeichnen mit dem Blitter sollten diese Bits alle auf 1 gesetzt sein.

BLTCON0 (040)

(BLiTer CONtrol register 0)

BLTCON1 (042)
(BLiTer CONtrol register 1)

Diese beiden Kontrollregister werden zusammen für die Steuerung der Blitter-Operationen benutzt. Jeweils einer der beiden Modi Area und Line wird durch das Bit 0 aus BLTCON1 aktiviert. Die folgende Tabelle ordnet den einzelnen Bits für die beiden Modi jeweils verschieden Namen zu. Die jeweilige Bedeutung eines gesetzten Bits entnehmen Sie bitte der darauffolgenden Beschreibung:

	Area-Modus		Line-Modus	
Bit	BLTCON0	BLTCON1	BLTCON0	BLTCON1
15	ASH3	BSH3	START3	TEXTURE3
14	ASH2	BSH2	START2	TEXTURE2
13	ASH1	BSH1	START1	TEXTURE1
12	ASA0	BSH0	START0	TEXTURE0
11	USEA	X	1	0
10	USEB	X	0	0
09	USEC	X	1	0
08	USED	X	1	0
07	LF7	X	LF7	0
06	LF6	X	LF6	SIGN
05	LF5	X	LF5	0
04	LF4	EFE	LF4	SUD
03	LF3	IFE	LF3	SUL
02	LF2	FCI	LF2	AUL
01	LF1	DESC	LF1	SING
00	LF0	LINE	LF0	LINE

Die wichtigen Bits im Area-Modus:

ASH3-0: Der "Shift"-Verschiebewert der Quelle A.
BSH3-0: Der "Shift"-Verschiebewert der Quelle B.
USEA-C: Modus-Kontrollbit für die Quellen A, B und C.
USED : Modus-Kontrollbit für das Ziel D.
LF7-0 : (Logic Function) Diese Bits enthalten das Minterm.
EFE : (Exclusive Fill Enable)
IFE : (Inclusive Fill Enable)
FCI : (Fill Carry Input)
DESC : (DEScending Control bit) Der Kopiervorgang läuft rückwärts.
LINE : Für den Area-Modus gelöscht.

Die wichtigen Bits im Line-Modus:

START3-0 : Diese Bits geben den Startpunkt der Linie an.
TEXTURE3-0:
LF7-0 : (Logic Function) Diese Bits enthalten das Minterm.
LINE : Für den Line-Modus gesetzt.

Die Bits SUD, SUL und AUL legen den Oktanten fest, indem die Linie gezeichnet wird:

Okt	SUD	SUL	AUL
0	1	1	0
1	0	0	1
2	0	1	1
3	1	1	1
4	1	0	1
5	0	1	0
6	0	0	0
7	1	0	0

BLTSIZE (058)

(BLiTter start and SIZE)

Im Area-Modus enthält dieses Register die Breite und die Höhe der Blitter-Operation. Im Line-Modus muß die Breite auf zwei gesetzt werden, die Höhe entspricht dann der Länge der Linie. Dabei enthalten die Bits 0 bis 5 die Breite in Worten, die Bits 6 bis 15 die Höhe (Länge) in Pixeln. ACHTUNG: wird dieses Register beschrieben, so startet der Blitter sofort mit seiner Operation, also dieses Register bitte als letztes beschreiben.

BLTxDAT (074)

(BLiTter source x DATa register)

Diese drei Register (das "x" steht für eine der Quellen A, B oder C) halten interne Daten für den Blitter bereit. Für den Line-Modus gilt noch folgendes: BLTADAT wird als Indexregister benutzt und muß mit 8000 initialisiert sein. BLTBDAT wird zur Musterung der Linie benutzt. Soll kein Linienmuster verwendet werden, muß BLTBDAT auf $FF gesetzt werden.

BLTxMOD (064)

(BLiTter MODulo)

Diese vier Register (das "x" steht für eine der Quellen A, B, C oder das Ziel D) enthalten den Modulo Wert, der am Ende einer Zeile automatisch zu dem jeweiligen Adreßzeiger hinzugezählt wird. Damit wird am Ende einer Zeile automatisch zu dem Anfang der nächsten gesprungen.

BLTxPTH (050)
(BLiTter PoinTer to x, High)
BLTxPTL (052)
(BLiTter PoinTer to x, Low)

Diese vier Registerpaare (das "x" steht für eine der Quellen A, B, C oder das Ziel D) enthalten jeweils den Low- und High-Anteil des Adreßzeigers auf den zu "x" gehörenden Speicherbereich. Diese Adreßzeiger sind jeweils 18 Bits lang, wobei in BLTxPTL die 15 Low-Bits und in BLTxPTH die übrigen drei High-

Bits stehen. Am Anfang der Blitteroperation zeigen diese Adreßzeiger auf das jeweils erste betroffene Wort, am Ende auf das jeweils zuletzt betroffene.

Literaturverzeichnis:

William M. Newman, Robert F. Sproull
Grundzüge der Interaktiven Computergrafik
McGraw-Hill Book Company GmBH

In diesem Buch werden die wesentlichen Themen der Computergrafik in Theorie und Praxis (PASCAL) behandelt.

Heinz-Otto Peitgen, Dietmar Saupe
The Science of Fractal Images
Springer-Verlag

Enthält nicht nur die Theorie über Berechnungen von fraktalen Grafiken, also auch L-Systeme und Landschaften, sondern neben zahlreichen Abbildungen auch die dazu notwendigen Algorithmen in einem gut verständlichen, Modula-2 Code.

Melvin L. Prueitt
Art and the Computer
McGraw-Hill Book Company

Ein Bildband mit fast 300 farbigen Abbildungen, die fast alle im Los Alamos National Laboratory entstanden. Es enthält leider nur wenig Hinweise über die Entstehung der Grafiken, ist jedoch dem ComputerGrafiker zur Inspiration durchaus zu empfehlen.

Amiga ROM Kernel Reference Manual: Includes & Autodocs Addison Wesley

Eines der englischen Standard-Werke zum Amiga, das nicht nur die dokumentierten Include-Files enthält, sondern auch sämtliche Systemroutinen beschreibt.

Amiga Hardware Reference Manual
Addison Wesley

Wer näheres über die Hardware der Coprozessor wissen möchte, findet in diesem Buch sämtliche Register der einzelnen Prozessoren gut dokumentiert.

Ernst A. Heinz
Amiga Basic Profibuch
Maxon

Nicht nur ein Buch für den BASIC-Programmierer, sondern für jeden, der in die Geheimnisse der Systemprogrammierung des Amigas eingeweiht werden möchte.

Wolf-Gideon Bleek, Bruno Jennrich, Peter Schulz
Amiga Intern Band 2
Data Becker

Für den "aktiven" Programmierer ein wichtiges Nachschlagewerk, vor allem wegen der ausführlichen Auflistung der Systemroutinen der Libraries und Devices.

